



University of Nebraska at Omaha
DigitalCommons@UNO

Student Work

5-2013

A MapReduce Algorithm for Finding Hotspots of Topics from Time Stamped Documents

Ashwathy Ashokan

University of Nebraska at Omaha

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>



Part of the [Computer Sciences Commons](#)

Recommended Citation

Ashokan, Ashwathy, "A MapReduce Algorithm for Finding Hotspots of Topics from Time Stamped Documents" (2013). *Student Work*. 2870.

<https://digitalcommons.unomaha.edu/studentwork/2870>

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.



A MapReduce Algorithm for Finding Hotspots of Topics from Time Stamped Documents.

A Thesis

Presented to the

Department of Computer Science and the Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Ashwathy Ashokan

May 2013

Supervisory Committee:

Dr. Parvathi Chundi, Chair

Dr. Sanjukta Bhowmick

Dr. Ilze Zigurs

UMI Number: 1535875

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI 1535875

Published by ProQuest LLC (2013). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

A MapReduce Algorithm for Finding Hotspots of Topics from Time Stamped Documents

Ashwathy Ashokan, MS
University of Nebraska, 2013
Advisor: Dr. Parvathi Chundi

Hotspots of a word/topic are time periods with a burst of activities in a time stamped document set. Identifying and analyzing hot spots of topics has been an important area of research. Finding hot spots of topics requires processing of contents of documents which is often time consuming. In this thesis, we explore MapReduce style algorithms for computing hot spots of topics. MapReduce is a distributed parallel programming model and an associated implementation for processing and analyzing large datasets. User specifies a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model and this thesis explores the feasibility of implementing the hotspot algorithm using MapReduce. We design map and reduce functions appropriate for preprocessing of documents, and the hot spot computation. We implement the functions in Hadoop (a MapReduce framework for Apache Foundation) and conduct several experiments to assess the benefits of MapReduce style implementation versus simple sequential implementation.

Keywords: Hotspots, MapReduce, Hadoop, Temporal Index

Dedication

This thesis is a dedication to my parents who are lifelong educators who taught me the importance of education and value of learning, for having presented me the opportunity for learning from some of the best possible institutions throughout my life. This Master's degree with the thesis-option was their inspiration and is a dream come true for them as much as it is for me.

I also dedicate this thesis to my husband for his unwavering patience, support and constant encouragement. He has been my inspiration as I navigated the obstacles in the completion of this research work.

Author's Acknowledgement

This thesis would not have been possible without the guidance and help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost, let me express my deepest gratitude to Dr. Parvathi Chundi, for her patience, steadfast encouragement and for her unfailing support as my thesis advisor. She has supported me through my thesis with her knowledge whilst allowing me the room to charter the unknown. One simply could not wish for a better or friendlier advisor. Other members of my supervisory committee, Dr. Ilze Zigurs and Dr. Sanjukta Bhowmick who have generously given their time and expertise to better my work.

Mr.Mohammad Shafiullah, our system administrator for working with me to fix environment and scheduling issues, even at insane hours.

Lastly but not the least, my parents for supporting me throughout my studies at the university, my 7 month old son for being patient while I was away working on my research, my husband for his inputs and guidance on the importance of demystifying the topics and presentations, my mother in law for her tips and guidance on thesis writing, my sister Anjali for helping me get started with Hadoop MapReduce and my best friend Amritha Balasubramaniam for her constant encouragement to sail through to the final destination.

Table of Contents

List of Figures-----	iv
Chapter 1 – Introduction-----	1
1.1 Problem Definition-----	2
1.2 MapReduce-----	3
1.3 Results-----	12
1.4 Future Work-----	13
Chapter 2 - Related work-----	14
Chapter 3 – MapReduce Applications-----	19
3.1 Word Count-----	19
3.2 Average of Integers-----	22
3.3 Natural Join-----	23
3.4 Augmenting Edges with Degrees in Graphs-----	26
3.5 Enumerating Triangles in Graphs-----	27
3.6 Enumerating Rectangles in Graphs-----	29
Chapter 4 – Hotspot Algorithm-----	33
4.1 Inverted Index -----	34
4.2 Time Point List-----	37
4.3 Time Point Index -----	39
4.4 Temporal Index-----	39
4.5 Hotspot Computation -----	43
Chapter 5 – Experiments-----	47
5.1 Dataset-----	47
5.2 Machine Configuration-----	48
5.3 Results-----	48
Chapter 6 - Conclusion -----	57
Bibliography-----	59

List of Figures

Figure 1.2.1 - Illustrates the MapReduce framework:	4
Figure 1.2.2 - The architecture of HDFS.	7
Figure 1.2.3 - MapReduce data flow with a single reducer task.....	9
Figure 1.2.4 - MapReduce data flow with multiple reduce tasks.....	10
Figure 1.2.5 - MapReduce data flow with no reduce tasks.	11
Figure 3.4.1 – Example illustration of augmenting edges with degree	27
Figure 3.4.2 – Example illustration of augmenting edges with degree.....	27
Figure 3.5.1 – Example illustration of enumerating triangles	29
Figure 3.5.2 – Example illustration of enumerating triangles.....	29
Figure 3.6.1 – Example illustration of enumerating rectangles.	31
Figure 3.6.2 – Example illustration of enumerating rectangles.	32
Figure 3.6.3 – Example illustration of enumerating rectangles	32
Figure 4.1 – High level block diagram of the thesis.....	33
Figure 4.1.1. Example document	34
Figure 4.1.2 - Block diagram of Inverted Index Algorithm.	34
Figure 4.1.3 Snapshot of Inverted Index	35
Figure 4.2.1 Block diagram of Time Point List Algorithm.	37
Figure 4.4.1- Example of InvertedIndex.txt	40
Figure 4.4.2- Example of TemporalIndex.txt	40
Figure 4.4.3- Example of TemporalIndex.txt	41
Figure 4.4.4- Block Diagram of Temporal Index Algorithm	41
Figure 4.5.1– Block Diagram of Hotspot Calculator Algorithm.....	43
Figure 5.3.2.1 – Table: Size of dataset	49
Figure 5.3.2.2 – Graph: Size of Inverted Index vs. Size of Temporal Index	50
Figure 5.3.3.1 – Table: # of distinct keywords in dataset	50
Figure 5.3.3.2 – Graph: Number of distinct keywords vs. Size of temporal index.....	51
Figure 5.3.4.1 – Table: Hotspot interval of some interesting keywords.	52
Figure 5.3.4.2 – Graph: Hotspot Timeline.....	53
Figure 5.3.5.1 –Table: MapReduce execution time	54
Figure 5.3.5.2 – Graph: MapReduce execution time vs. Size of Temporal Index.....	54
Figure 5.3.5.3 – Graph: Rate of change of MapReduce execution time vs. Size of Temporal Index	55
Figure 5.3.6.1 – Table: MapReduce and Sequential execution time for the keyword “yahoo”.....	56
Figure 5.3.6.2 – Graph: Sequential Execution time vs. MapReduce Execution time for the keyword “yahoo”.....	56

1. Introduction

Analyzing unstructured text documents such as blogs, news articles etc. for temporal information is an important data mining activity due to the pervasive nature of these data [3]. It is common for blogs and news sources to discuss/publish a few news stories intensely for a period of time. The topics covered in news stories may change frequently and be replaced with new topics, or they may stay active and the context surrounding the topic may change. This kind of coverage results in bursty patterns of stories/topics. It has been well-recognized that identifying periods of bursty activity of a topic may provide a lot of useful information [3, 5, 16] that could be utilized by businesses, policy makers, and researchers. Extracting the **hotspot** of topics in a time-stamped document set is one of the ways for identifying and analyzing such bursty patterns. The time periods of these bursty patterns for a particular topic/word are identified, and the time period of maximum occurrence of the topic/word is known as the *hotspot* for the topic/word. Methods such a term or document frequency can be used to compute the presence of a topic in a document set. Each interval in the time period of the document set is associated with a numeric value which we call the *discrepancy score*. A high discrepancy score indicates that the documents in the time interval are more focused on the topic than those outside of the time interval. A *hot spot* of a given topic is defined as a time interval with a highest discrepancy score.

The naïve implementation for extracting hot spot of a topic is a very expensive algorithm with a running time of $O(n^3)$ and this is especially problematic for large

datasets such as blogs. In this thesis, we explore the MapReduce style programming to see if it would make the naïve implementation of hot spot extraction more efficient.

1.1 Problem definition

A *hotspot* of a topic in a given data set of time stamped documents is a subinterval of the time period which contains significantly more documents that discuss the topic than the rest of the time period. Identifying hot spot may provide a lot of useful information.

To identify hot spots, we assign a discrepancy score to each of the $O(n^2)$ intervals of the time period of the document set. A discrepancy score of an interval is a numerical value that captures the discrepancy between the presence of the topic in the document set of the interval and its presence in the document set outside the interval. We use the temporal scan statistic to compute the discrepancy score of an interval [17, 18].

We define the hot spot extraction problem as following: given a time stamped document set and a topic, identifies a time interval with the maximum discrepancy score. Note, there may be more than one such intervals; we arbitrarily choose one of those intervals as a hot spot. Extracting a hot spot requires calculating the discrepancy score of every interval in the time period of the document set. A naïve implementation runs in time $O(n^3)$, where n is the number of the time points of the document set. This paper discusses using the MapReduce style programming to

improve the efficiency of the naïve implementation and present a MapReduce algorithm to compute hot spots.

1.2 MapReduce

MapReduce is a programming model and an associated implementation for processing and generating large data sets. It is a distributed, parallel, fault-tolerant and scalable programming model. Today, it is largely being used for expressing distributed computations on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity servers. . It was originally developed by Google and built on well-known principles in parallel and distributed processing dating back several decades. MapReduce has since enjoyed widespread adoption via an open-source implementation called Hadoop, whose development was led by Yahoo (now an Apache project). Today, a vibrant software ecosystem has sprung up around Hadoop, with significant activity in both industry and academia [6].

MapReduce builds on the observation that many information processing tasks have the same basic structure: a computation is applied over a large number of records (e.g., Web pages) to generate partial results, which are then aggregated in some fashion. Naturally, the per-record computation and aggregation function vary according to task, but the basic structure remains fixed. Taking inspiration from higher-order functions in functional programming, MapReduce provides an abstraction at the point of these two operations. Specifically, the programmer defines a “mapper” and a “reducer” with the following signatures:

$$\text{map}: (k1, v1) \rightarrow [(k2, v2)]$$
$$\text{reduce}: (k2, [v2]) \rightarrow [(k3, v3)]$$

Key/value pairs form the basic data structure in MapReduce. The mapper is applied to every input

key/value pair to generate an arbitrary number of intermediate key/value pairs. The reducer is applied to all values associated with the same intermediate key to generate output key/value pairs. This two-stage processing structure is illustrated in Figure 1 [9].

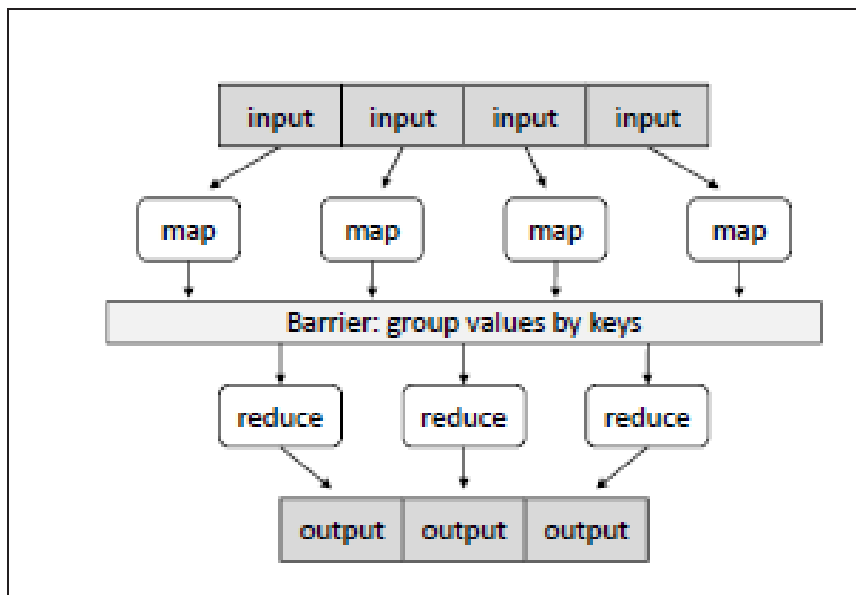


Figure 1.2.1 - Illustrates the MapReduce framework: the “mapper” is applied to all input records, which generates results that are aggregated by the “reducer”

Under the framework, a programmer need only provide implementations of the mapper and reducer.

On top of a distributed file system [4], the runtime transparently handles all other aspects of execution, on clusters ranging from a few to a few thousand nodes. The runtime is responsible for scheduling map and reduce workers on commodity hardware assumed to be unreliable, and thus is tolerant to various faults through a number of error recovery mechanisms. The runtime also manages data distribution, including splitting the input across multiple map workers and the potentially very large sorting problem between the map and reduce phases whereby intermediate key/value pairs must be grouped by key [9].

MapReduce allows for distributed processing of the map and reduction operations. Provided each mapping operation is independent of the others, all maps can be performed in parallel – though in practice it is limited by the number of independent data sources and/or the number of CPUs near each source. Similarly, a set of 'reducers' can perform the reduction phase - provided all outputs of the map operation that share the same key are presented to the same reducer at the same time, or if the reduction function is associative. While this process can often appear inefficient compared to algorithms that are more sequential, MapReduce can be applied to significantly larger datasets than "commodity" servers can handle – a large server farm can use MapReduce to sort a petabyte of data in only a few hours. The parallelism also offers some possibility of recovering from partial failure of servers or storage during the operation: if one mapper or reducer fails, the work can be rescheduled – assuming the input data is still available [1].

Hotspot computation usually deals with huge chunks of text data and the scalability of the MapReduce programming model might be the answer to making its extraction process faster and more efficient. We devote this thesis to further investigate if MapReduce programming model makes sense for the Hotspot extraction problem by implementing the naïve implementation (sequential implementation) of the Hotspot extraction mentioned in the reference [*Wei Chen, Parvathi Chundi*] in the MapReduce framework provided by Hadoop.

Hadoop

The Hadoop implementation of MapReduce and is primarily written in Java. However, it provides methods of writing the core parts of a job in other languages, as long as they support streams, such as C++ and Python. Hadoop has also introduced two higher level abstractions from MapReduce, called Pig and Hive. Pig provides a scripting language which can describe a MapReduce job. Hive was developed by Facebook and it implements an SQL like language on-top of MapReduce. Both of these projects provide a simplified method of implementing a job, hiding the details of dealing with MapReduce. They are also considered the best way to implement more complicated logic such as joins, which users of relational databases take for granted, but in the MapReduce world are much harder to code [7].

HDFS is a distributed, scalable, and portable file system written in Java for the Hadoop framework. Each node in a Hadoop instance typically has a single namenode; a cluster of datanodes form the HDFS cluster. The situation is typical because each

node does not require a datanode to be present. Each datanode serves up blocks of data over the network using a block protocol specific to HDFS. The file system uses the TCP/IP layer for communication. Clients use RPC(Remote procedure call) to communicate between each other. HDFS stores large files (an ideal file size is a multiple of 64MB), across multiple machines. It achieves reliability by replicating the data across multiple hosts, and hence does not require RAID (Redundant Array of Inexpensive Disks) storage on hosts. With the default replication value, 3, data is stored on three nodes: two on the same rack, and one on a different rack. Data nodes can talk to each other to rebalance data, to move copies around, and to keep the replication of data high. HDFS is not fully POSIX compliant, because the requirements for a POSIX file system differ from the target goals for a Hadoop application. The tradeoff of not having a fully POSIX-compliant file system is increased performance for data throughput. HDFS was designed to handle very large files [14].

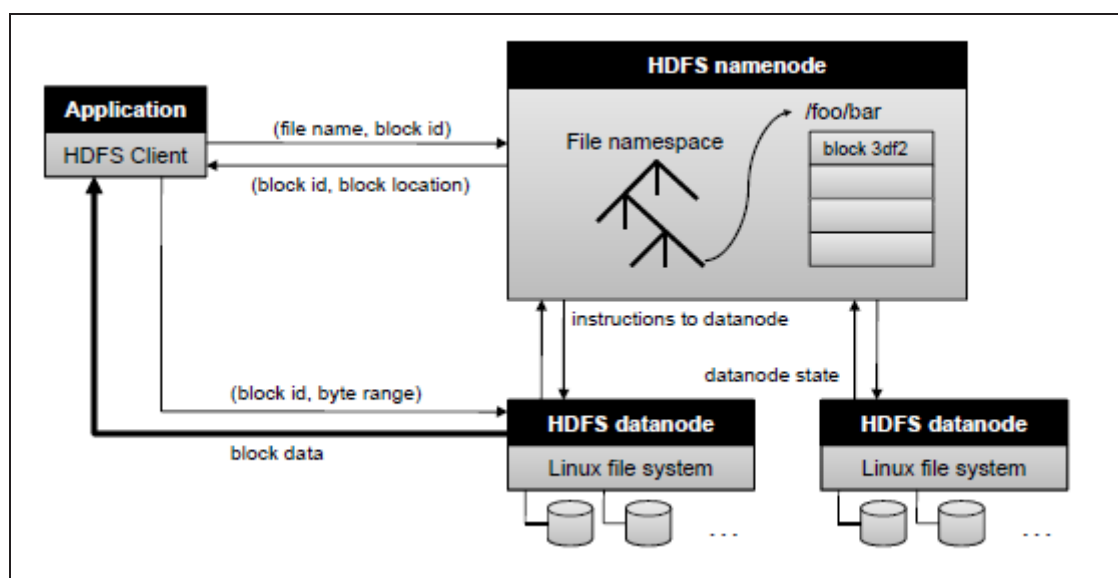


Figure 1.2.2 - The architecture of HDFS. The namenode (master) is responsible for maintaining the file namespace and directing clients to datanode (slaves) that actually hold data blocks containing user data.

A MapReduce job: Deeper look

A MapReduce job is a unit of work that the client wants to be performed: it consists of the input data, the MapReduce program, and configuration information. Hadoop runs the job by dividing it into tasks, of which there are two types: map tasks and reduce tasks.[2]

There are two types of nodes that control the job execution process: a jobtracker and a number of tasktrackers. The jobtracker coordinates all the jobs run on the system by scheduling tasks to run on tasktrackers. Tasktrackers run tasks and send progress reports to the jobtracker, which keeps a record of the overall progress of each job. If a task fails, the jobtracker can reschedule it on a different tasktracker.[2]

Hadoop divides the input to a MapReduce job into fixed-size pieces called input splits, or just splits. Hadoop creates one map task for each split, which runs the user defined map function for each record in the split. For most jobs, a good split size tends to be the size of a HDFS block, 64 MB by default, although this can be changed for the cluster (for all newly created files), or specified when each file is created. Hadoop does its best to run the map task on a node where the input data resides in HDFS. This is called the data locality optimization [2].

Map tasks write their output to local disk, not to HDFS. This is because the Map output is intermediate output: it's processed by reduce tasks to produce the final output, and once the job is complete the map output can be thrown away. So storing it in HDFS, with replication, would be overkill. If the node running the map task fails before the map output has been consumed by the reduce task, then Hadoop will automatically rerun the map task on another node to recreate the map output [2].

Reduce tasks don't have the advantage of data locality—the input to a single reduce task is normally the output from all mappers. In most of the cases, we have a single reduce task that is fed by all of the map tasks. Therefore the sorted map outputs have to be transferred across the network to the node where the reduce task is running, where they are merged and then passed to the user-defined reduce function. The output of the reducer is normally stored in HDFS for reliability. For each HDFS block of the reduce output, the first replica is stored on the local node, with other replicas being stored on off-rack nodes. Thus, writing the reduce output does consume network bandwidth, but only as much as a normal HDFS write pipeline consumes [2].

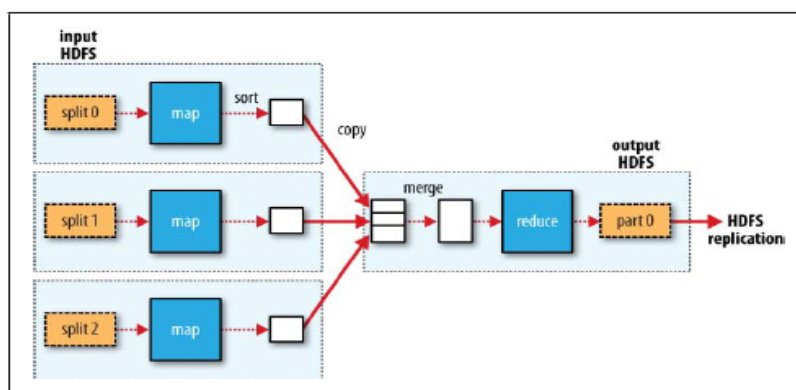


Figure 1.2.3 - MapReduce data flow with a single reducer task

The whole data flow with a single reduce task is illustrated in figure below. The dotted boxes indicate nodes, the light arrows show data transfers on a node, and the heavy arrows show data transfers between nodes. [2]

The number of reduce tasks is not governed by the size of the input, but is specified independently. When there are multiple reducers, the map tasks partition their output, each creating one partition for each reduce task. There can be many keys (and their associated values) in each partition, but the records for every key are all in a single partition. The partitioning can be controlled by a user-defined partitioning function, but normally the default partitioner—which buckets keys using a hash function—works very well [2].

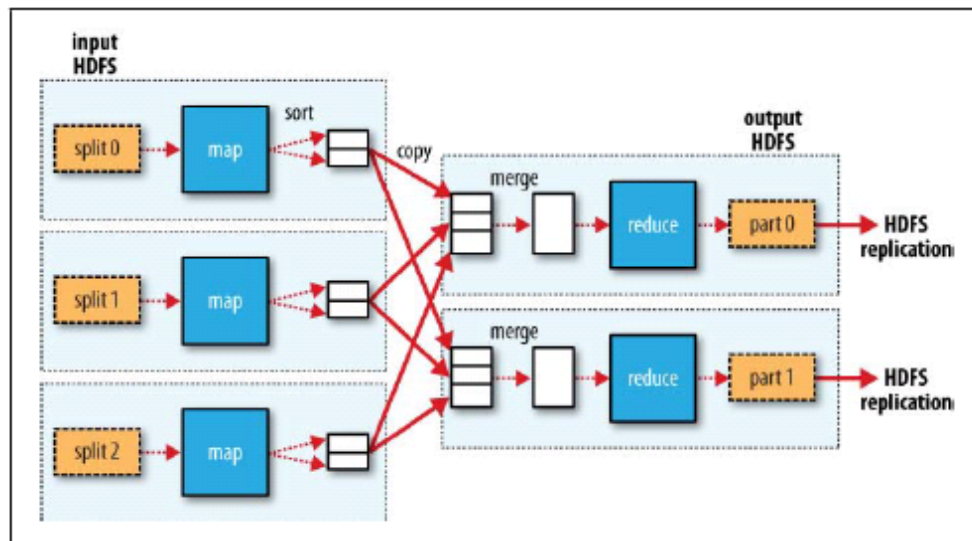


Figure 1.2.4 - MapReduce data flow with multiple reduce tasks.

The data flow for the case of multiple reduce tasks is illustrated in figure below. This diagram makes it clear why the data flow between map and reduce tasks is

colloquially known as “the shuffle,” as each reduce task is fed by many map tasks. The shuffle is more complicated than this diagram suggests, and tuning it can have a big impact on job execution time.[2]

Finally, it’s also possible to have zero reduce tasks. This can be appropriate when you don’t need the shuffle since the processing can be carried out entirely in parallel. In this case, the only off-node data transfer is when the map tasks write to HDFS. (see figure below).[2]

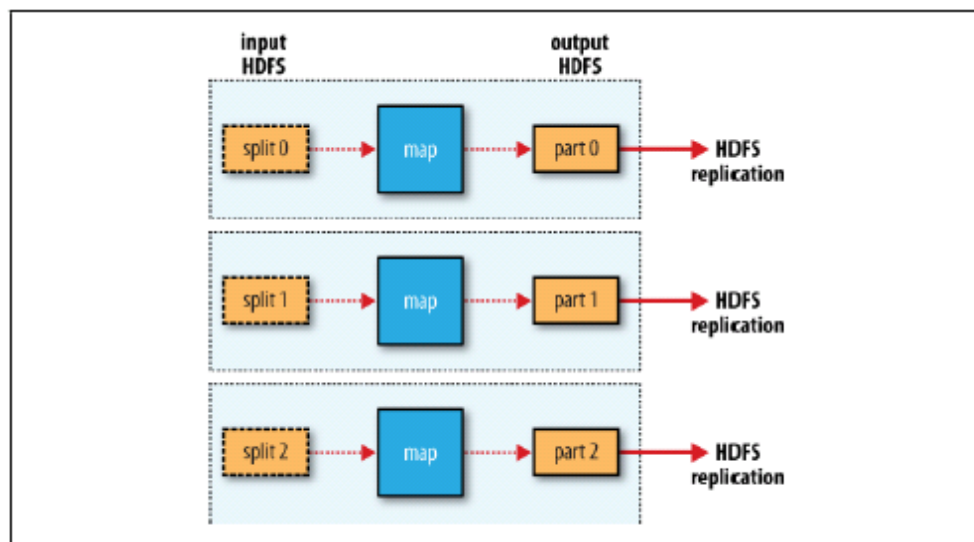


Figure 1.2.5 - MapReduce data flow with no reduce tasks.

Many MapReduce jobs are limited by the bandwidth available on the cluster, so it pays to minimize the data transferred between map and reduce tasks. Hadoop allows the user to specify a combiner function to be run on the map output—the combiner function’s output forms the input to the reduce function. Since the combiner function is an optimization, Hadoop does not provide a guarantee of how many times it will call

it for a particular map output record, if at all. In other words, calling the combiner function zero, one, or many times should produce the same output from the reducer.[2]

1.3 Results

The naïve hotspot algorithm was implemented using two methodologies for the comparison of efficiency

1. Sequential Algorithm (Stand-alone Java Application), and
2. MapReduce Algorithm (Hadoop Framework)

An eight node cluster was used for execution. The dataset consisted of about 20,000 files of blog data (about 200MB size) from the time period 1st August 2008 to 30th September 2008.

Each of the algorithms was run a number of times on these datasets, incrementally increasing their time span and data size to make a study on their response times.

It was observed that as the data set increased in size, MapReduce algorithm was exponentially faster than the sequential algorithm. For the largest data set size tested (200MB), MapReduce algorithm was found to reduce execution time by about 2.64 times compared to Sequential Algorithm. It was also noted that Map Reduce algorithm was inefficient for smaller datasets owing to the higher overhead related to distributed implementation.

1.4 Future Work

We plan to run the algorithm on larger dataset to perform a more detailed study on the scalability of the hot spot algorithm using Map Reduce as it was found to perform much better with larger datasets.

It was observed that the mapper of the Map Reduce program took significantly more time than the reducer. We attribute its behavior to the mapper running doing all of the discrepancy score calculation which is a $O(n^3)$ algorithm while the reducer just calculating the maximum of all the discrepancy scores calculated. We plan to redesign this algorithm to balance out the work done by the mapper and reducer to improve scalability.

There exists an improved version of the native hotspot algorithm called the Efficient Hotspot Extraction [3], and we plan to implement that in the Map Reduce framework for an even better scalability.

Finally, we would like to extend the Map Reduce algorithms to compute hotspots for streaming data.

2. Related work

Identifying time periods with a burst of activities related to a topic has been an important problem in analyzing time stamped documents. Extracting the hot spot of a given topic in a time-stamped document set is one of the key interests of text miners [3]. The paper talks about hot spot extraction on both basic topics, containing a simple list of keywords and complex topics, containing keyword connected with the logical operators AND, OR and NOT. A concept of *measure* based on the Fuzzy Set Theory to compute the amount of information related to the topic in a document set. It also introduces the notion of a *topic DAG* to facilitate an efficient computation of measures of complex topics. The inspiration for this thesis came from this paper which discusses the hotspot extraction problem and its naïve implementation with a run time of $O(n^3)$. It then constructs a more efficient version (EHE) which has a run time of $O(n^2)$. There has been little work done in exploring the usage of data-intensive computing frameworks like MapReduce for such problems. As there exists no hot spot computation algorithms on MapReduce we have taken the naïve implementation of the Hotspot Extraction Algorithm and implemented it in the MapReduce framework to analyze its performance. Several experiments were conducted and it was shown that the EHE algorithm outperformed the naïve one significantly and the extracted hotspots of given topics were meaningful.

“Necessity is the mother of invention”, and such was the case with MapReduce. For years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled

documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues [4]. And thus MapReduce was born!

MapReduce allowed the expression of the simple computations involved but hide the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs. MapReduce provides a fault-tolerant implementation that scales to thousands of processors while most of the parallel processing systems have only been implemented on smaller scales and leave the details of handling machine failures to the programmer [4].

MapReduce has been a huge success at Google due to its ease of use, even for programmers without experience with parallel and distributed systems as it hides the details of parallelization, fault-tolerance, locality optimization, and load balancing.

Additionally a large variety of problems are easily expressible as MapReduce computations such as for sorting, data mining, machine learning etc. Lastly, MapReduce jobs were found to be highly scalable [4]. We took our inspiration from this paper when we decided to implement the Hotspot algorithm, a text mining algorithm in this framework.

MapReduce is being explored as a solution to the scalability issue by different spheres of computation that deals with huge chunks of data like bioinformatics, web, any system that deals with geographic data, social networking graph data etc. Attempts have been made to implement a lot of existing algorithms in the MapReduce framework in an attempt to increase scalability. These MapReduce algorithms are not obviously analogs of standard algorithms and for the most part require a complete rethinking of the problem.

As the size of graphs for analysis continues to grow, methods of graph processing that scale well have become increasingly important. One way to handle large datasets is to disperse them across an array of networked computers, each of which implements simple sorting and accumulating, or MapReduce operations. The reference [*Jonathan Cohen*] talks about the possibility of considering cloud computing for graph operation if they can be decomposed into logical steps that fit the MapReduce cycle. This in addition offers a way to handle a large graph on a single machine that cannot hold the entire graph as well as the possibility of enabling streaming graph processing. This paper gives a list of graph operation which the author thinks might be feasible to

implement in the MapReduce framework. Specifically, the author these are some of the graph operation that author thinks are feasible to be implemented in the MapReduce framework though no information on the implementation details has been provided in the paper :- Augmenting Edges with Degrees, Simplifying the Graph, Enumerating Triangles, Enumerating Rectangles, Finding Trusses, Barycentric Clustering and Finding Components. Actually, some of them are very easy problems if they can traverse graphs. However, as the author mentions, traversing graphs with MapReduce is very inefficient since a mapper reads only a record randomly for each map operation. Hence, all the operations that the paper proposed avoid traversing graphs. Instead, their common pattern in graph algorithms proposed has at least two MapReduce programs line up together as follows:

- A map operation: Read and process all the edges (or vertex) or changing some piece of edge (or vertex) information. Then, result in records by vertex as key.
- A reduce operation: For each record obtained from the previous map operation, read and determine the updated state of vertex or edge; emit this information in partially (or locally) updated records. Then, results in them.
- A map operation: Identity mapper – Mapper that just read the input and emits it to the reducer without any processing.
- A reduce operation: For each record from the previous reduce operation, combine the updates globally and complete updated information.

As a part of my research, I have developed the algorithm for a couple of these graph operations, the details of which have been discussed in Chapter 4. In the end the

author concludes that not all algorithms can be implemented in the MapReduce framework and not all algorithms make sense to be implemented in this framework because sometimes it is implementable but ends up being impractical and inappropriate. Our experiment will tell us if the Hotspot Algorithm is a fit for the MapReduce framework.

The growth of the internet has pushed researchers from all disciplines to deal with volumes of information where the only viable path is to utilize data-intensive frameworks. Genetic algorithms are increasingly being used for large scale problems like non-linear optimization, clustering and job scheduling. The traditional MPI-based parallel GAs requires detailed knowledge about machine architecture. Reference [16] demonstrate a transformation of genetic algorithms into the map and reduce primitives and implement the MapReduce program and demonstrate its scalability to large (10^5) problem sizes.

3. MapReduce Applications

Transformation of an existing algorithm MapReduce programming, a lot of times require complete re-thinking of the problem as the MapReduce algorithms are not obvious analogs of standard algorithms. This chapter is devoted to providing illustration on how some of the key algorithms that have real world applications can be modeled into a MapReduce one.

Let us start with a very simple program for counting the number of words and their frequencies in a document which has its applications in Log Analysis and Data Querying.

3.1 Word Count

Input: Document collection

Output: word and its total frequency across the document collection

Sequential Implementation

Input: Document Collection

Output: Hashmap WordCount with Words as keys and their frequency across document set as values.

```
Class SequentialWordCount
{
    main()
    1. Define HashMap<String,Integer> WordCount.
    2. Go through files in the document set.
        a. Tokenize the files
        b. For each word
```

```

        i. If the word does not exist in WordCount
            Insert <word,1>
        ii. Else
            Update the record in WordCount as <word,freq
            + 1>
    }

```

MapReduce Implementation

Input: Document Collection

Output: File with the information <word,freq across the document set>

```

Class MapReduceWordCount
{
    map(input files)
    {
        1. for each file
            a. Tokenize the file
            b. Emit(word, 1)
    }
    reduce(key,[values])
    {
        1. for val: values
            a. sum += val
        2. emit(key,sum)
    }
}

```

The mapper tokenizes the file and emits each word as the key and the integer 1 as value. This becomes the input to the reducer. But before entering the reduce the data emitted from the mapper goes through a shuffle and sort phase where the data is sorted by key and the values for each key is aggregated into a list. Hence the reducer gets <word,[1,1,1...]> as the input and all the 1's get added up into the variable sum and then get emitted as the final output. As you can see the MapReduce framework take care of the logic behind grouping all the occurrence of a word and all we need to do is just add it up while in the sequential version we have to keep track of it.

Example Illustration:

Suppose the document collection is represented as the following:-

<document id, document text> pairs.

Input to the Mapper:-

```
<0, "facebook makes deal">
<1, "how facebook could">
<2, "reports of verizon">
<3, "apple google mobile">
<4, "verizon iphone apple">
<5, "twitter ad revenue">
```

Output of Mapper and input to the Shuffle and Sort phase:-

```
<facebook,1>,<makes,1>,<deal,1>,
<how,1>,<facebook,1>,<could,1>
<reports,1>,<of,1>,<Verizon,1>
<apple,1>,<google,1>,<mobile,1>
<verizon,1>,<iphone,1>,<apple,1>
<twitter,1>,<ad,1>,<revenue,1>
```

Output of Shuffle and Sort and Input to the Reducer:

```
<ad, [1]>
<apple, [1,1]>
<could, [1]>
<deal, [1]>
<facebook,[1,1]>
<google, [1]>
<how, [1]>
<iphone, [1]>
<makes, [1]>
<mobile, [1]>
<of, [1]>
<reports, [1]>
<revenue,[1]>
<twitter, [1]>
<verizon,[1,1]>
```

Output of the Reducer:

```
<ad, 1>
<apple, 2>
```

```

<could, 1>
<deal, 1>
<facebook,2>
<google, 1>
<how, 1>
<iphone,1>
<makes, 1>
<mobile, 1>
<of, 1>
<reports,1>
<revenue,1>
<twitter, 1>
<verizon,2>

```

3.2 Average of Integers

Input: Large file with a list of integers

Output: Average of all integers

Sequential Implementation:

Input: Large file with integers

Output: Average of all integers

```

Class SequentialAverageOfIntegers
{
    main()
    {
        for each line in file
            for each integer i in file
                sum = sum + i
                count = count + 1
            avg = sum/count;
        }
    }
}

```

MapReduce Implementation of Average of Integers

```

Class MapReduceAverageOfIntegers
{

```

```

map(input file)
{
    for each line in file
        for each integer i in file
            emit(any_key, i)
}

reduce(any_key, list of integers)
{
    for i: list of integers
    {
        sum = sum + i;
        count = length of list of integers;
    }
    avg = sum/count;
    emit(any_key, avg);
}
}

```

In case of the above example, we don't care about the key the mapper emits as we need all the numbers together in the reducer to produce the average since the shuffle and sort phase groups the value by the key we emit all the integers with the same key. As you can see the remodeling the program to the MapReduce framework did not really any extra efficiency as the reducer is doing exactly what the sequential version of it does. This example is just an example of a program that can be modeled as a MapReduce program but isn't a good fit to be a MapReduce program.

3.3 Natural Join

Input: 2 files with 2 tables with a common attribute

Output: Join of 2 tables

Sequential Implementation

Class SequentialNaturalJoin

```

{
    main()
    {
        1. Open file1
        2. Open file2
        3. For every line in file 1
            a. For every line in file 2
                i. If they common attribute value matches
                    Write all the attributes from both
                    the files to the output file
            }
        }
    }
}

```

MapReduce Implementation

Input: a directory containing both the files, each containing the table that has to be joined

Output: a file with the join of 2 files

Class MapReduceNaturalJoin

```

{
    map(2 files contains the tables to be joined)
    {
        For each file f
            For each line in the file (each record)
                Emit(join_attr, f + "-" +
                    remaining_attr_of_the_rec)
        }

        reduce(join_attr, [List of f-
remaining_attr_of_the_rec])
        {
            for r1: 1 to f-remaining_attr_of_the_rec
            {
                for r2: 2 to f-remaining_attr_of_the_rec
                {
                    Extract f from r1
                    if r2 does not contain f
                        record = join_attribute + (r1-f)
+ (r2-f)
                        Emit(record, - );
                }
            }
        }
    }
}

```


The output file contains the natural join of the tables in both the input files

Example Illustration:

File1 contains the attributes Emp ID, Name:

<111, Jim>

<222, Joy>

<333, Ryan>

File2 contains the attributes EmpID, DeptName:

<111, MIS>

<111, CS>

<222, Physics>

Input to Mapper:

The above will be the input to Mapper.

Output of Mapper:

(111, file1-Jim), (222, file1-Joy), (333, file1-Ryan), (111, file2-MIS), (111, file2-CS),
(222, file2-Physics)

This will be the input to shuffle and sort phase

Output of Shuffle and Sort:

(111,[file1-Jim,file2-MIS,file2-CS])

(222,[file1-Joy,file2-Physics])

(333,[file1-Ryan])

This will be the input to the reducer.

Output of Reducer:

(111,Jim,MIS)

(111,Jim,CS)

(222,Joy,Physics)

As you can see, the MapReduce version of join algorithm is no better than the sequential version as the reducer still has a run time of $O(n^2)$. We have very intelligent and powerful tools handy in Hadoop like Hive and Pig that can easily join huge data sets with the choice of join like inner, outer etc.

Sections 3.4, 3.5 and 3.6 illustrate examples based on graph operations that have been implemented based on the reference [10]. The paper mentions a list of possible graph operation that can possibly be implemented in the MapReduce framework. The paper gives an example illustration of 3.4 and no algorithm as such and nothing really on the 3.5 and 3.6. As a part of my research I have developed the algorithms illustrated in sections 3.5 and 3.6..

3.4 Augmenting Edges with Degrees in Graphs

These algorithms contain a series of MapReduce jobs.

Input: edges.

Output: edges augmented with the degree of each of its vertices.

```

Class MapReduceAugEdgesWithDegree1
{
    map1(key, edge e)
    {
        for each vertex v in e
            emit(v, e)
    }
    reduce1(v, [e1 , e2 ..... en])
    {
        d = sizeOf([e1 , e2 ..... en])
        for each edge e in [e1 , e2 ..... en]
            emit(e, d(v))
    }
}

Class MapReduceAugEdgesWithDegree2
{
    map2(e, d(v))
    {
        emit(e, d(v))
    }
    reduce2(e, [d(v), d(v')])
    {

```

```

    emit(e, (d(v), d(v')))
  }
}

```

Example Illustration

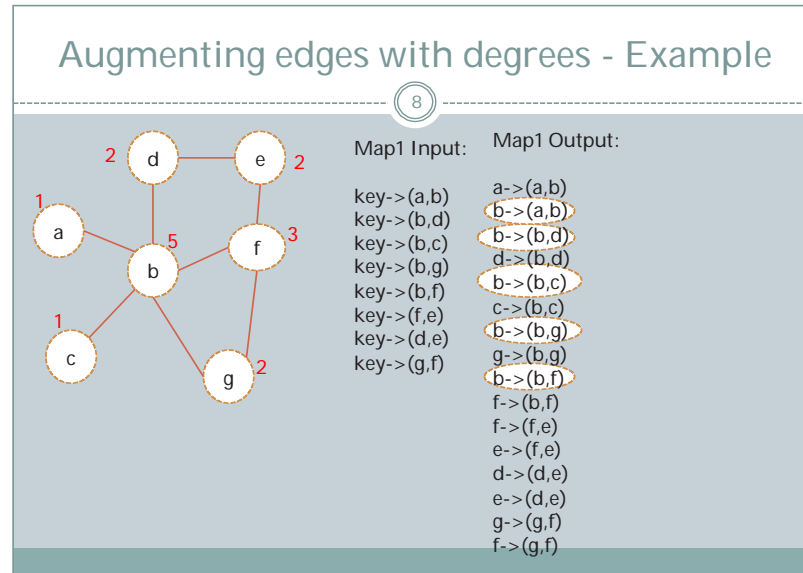


Figure 3.4.1 – Example illustration of augmenting edges with degree mapper 1 input and output.

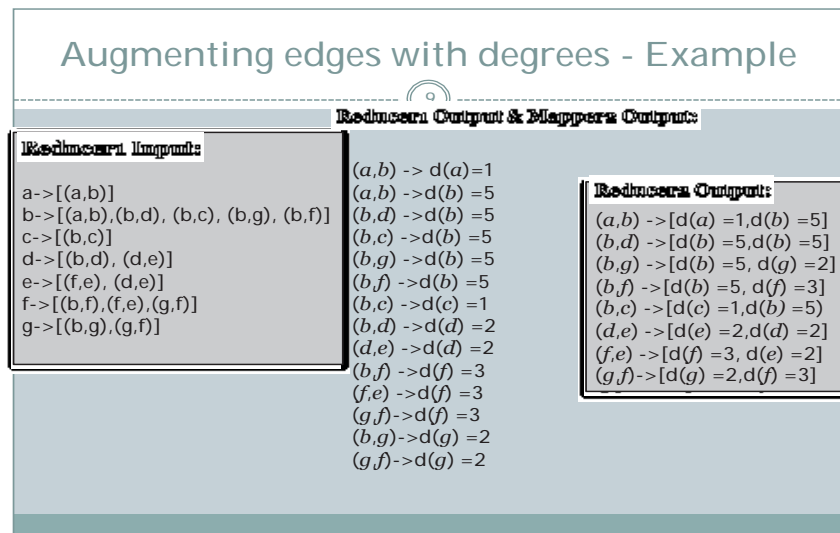


Figure 3.4.2 – Example illustration of augmenting edges with degree reducer 1, mapper 2 and reducer 2 input and output.

3.5 Enumerating Triangles in Graphs

Enumerating triangles is essentially a two-step approach: enumerate open triads (pairs of edges of the form $\{(A, B), (B, C)\}$) and recognize when an edge closes those triads to form triangles. To find triangles, I can choose a vertex ordering, bin all edges under their minimum vertex, and test each pair of edges recorded in each bin to see if that pair (forming an open triad) is closed by a third edge.

Input: edge list

Output: list of edges forming a triangle

This requires 2 sets of MapReduce jobs executed in sequence. First MapReduce job finds the open triads and bins it under its closing edge. Second MapReduce job looks in the edge list to see if the closing edge exists and if yes emits the closed triads.

```

Class MapreduceEnumeratingTriangles1
{
    map1(key, edge e)
    {
        for each edge  $e = (v, v')$ 
            if( $d(v) < d(v')$ )
                emit( $v, e$ )
            else
                emit( $v', e$ )
    }
    reduce1( $v, [e_1, e_2, \dots, e_n]$ )
    {
        go through the list of edges to find open triads.
        when every a pair of open triads are found
            emit(its closing edge  $e_c$ , open triad pair)
    }
}

```

Note:- we create a separate input file for map2 which contains the (closing edge e_c , open triad) pair appended to the (edge, edge) pair

```

Class MapReduceEnumeratingTriangles2
{
    map2(edge e, edge list)
    {
        emit(e, edge list)
    }
    reduce2(e, list of edge list)
    {
        if(sizeof(flatten(list of edge list)) == 3)
            emit(e, flatten(list of edge list))
    }
}

```

Example Illustration

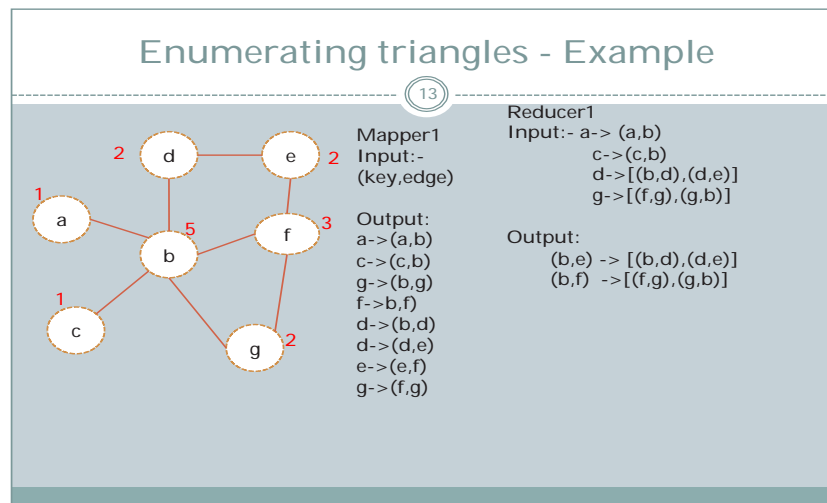


Figure 3.5.1 – Example illustration of enumerating triangles, mapper 1 and reducer 1 input and output.

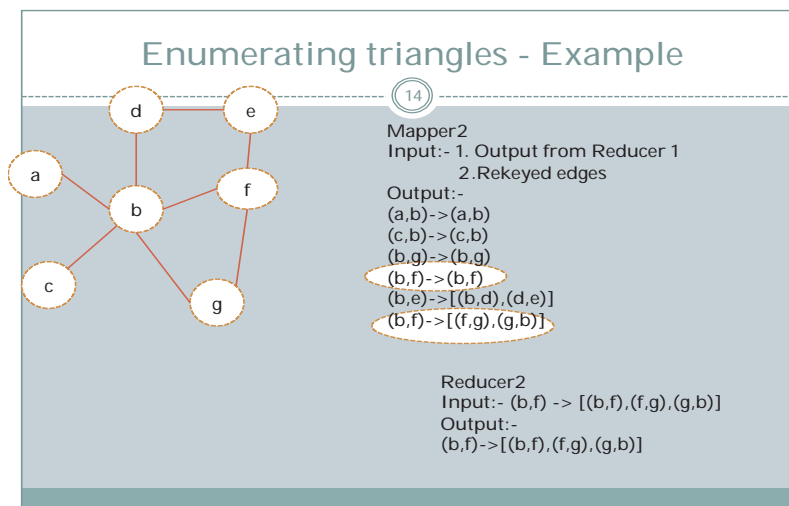


Figure 3.5.2 – Example illustration of enumerating triangles, mapper 2 and reducer 2 input and output.

3.6 Enumerating Rectangle in Graphs

The job of enumerating rectangles (4-cycles) is similar to that of enumerating triangles. Here, the approach is to find two open triads connecting the same pair of vertices; their combination is a rectangle.

Input: edge list

Output: edge list of a rectangle

The general logic is to find triads connecting the same pair of vertices. This also requires 2 sets of MapReduce jobs executed in sequence. First MapReduce job finds all the triads in the graph. Second MapReduce job groups the triads connecting same pair of vertices, ie, with the same closing edge.

```

Class MapReduceEnumeratingRectangles1
{
    map1(key,edge e)
    {
        for each edge e = (v, v')
            if(d(v) < d(v'))
                emit(v,(e,e.low))
                emit(v',(e,e.high))
            else
                emit(v',e.low)
                emit(v,e.high)
    }

    reduce1(vertex v, edge[e1.order, e2.order..... en.order])
    {
        for each edge ei in the edge list
            if(ei.order == low)
            {
                for each edge ej in the edge list
                    if(ei != ej)
                    {
                        if( ei and ej are open triads)

```

```

        emit(closing edge of  $e_i$  and  $e_j$  ,  $(e_i, e_j)$ 
    )
    }
}
}

```

```

Class MapReduceEnumeratingRectangles2
{
    map2(closing edge, open triad)
    {
        emit(closing edge, open triad)
    }

    reduce2(closing edge, list of open triads)
    {
        emit(key, triad pair)
    }
}

```

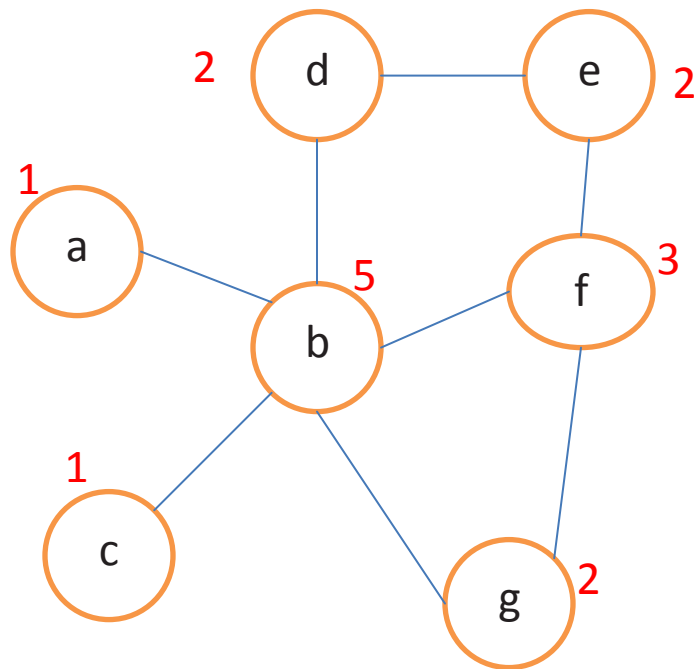


Figure 3.6.1 – Consider the above graph for the example illustration

Example Illustration

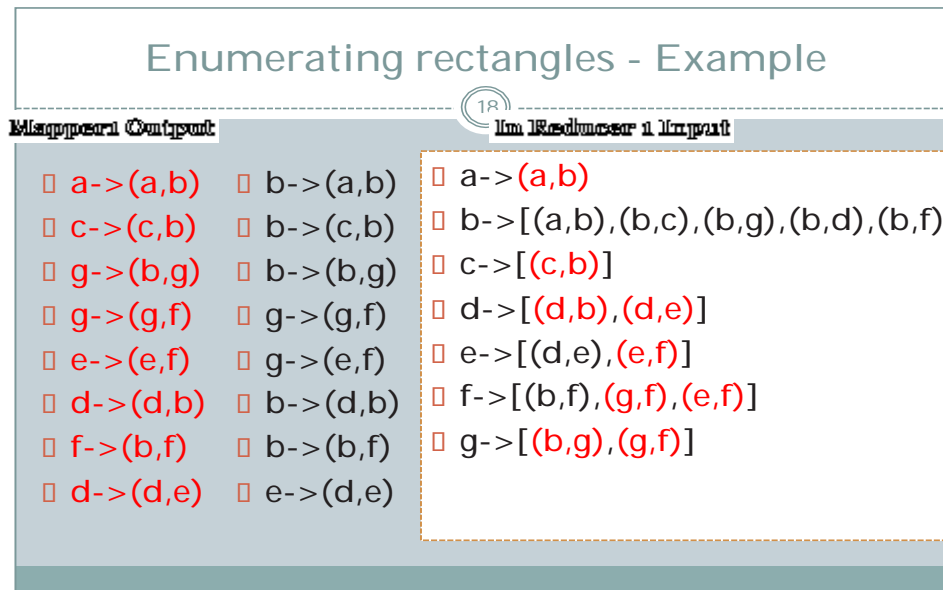


Figure 3.6.2 – Example illustration of enumerating rectangles, mapper 1 and reducer 1 input and output.

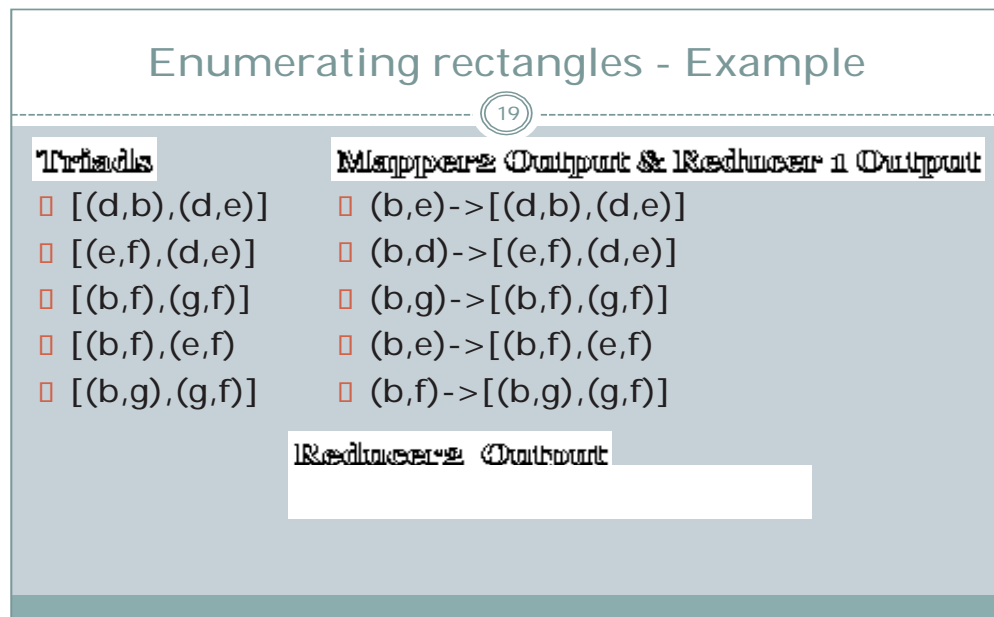


Figure 3.6.3 – Example illustration of enumerating rectangles, mapper 2 and reducer 2 input and output.

4. Hot Spot Algorithm

This section discusses all the algorithms involved in this experiment. This experiment involved taking the raw blog data from Spinn3r and pre-processing them first to generate the Inverted Index, Time Point Index and then Temporal Index, before running the hot spot extraction algorithm on it.

Below is a block diagram that gives a high level overview of all the different processing algorithms, and the flow of processed data through them. This project at a high level takes the path to the dataset and a topic(query word) as input and then returns the hot spot interval of the topic(query word).

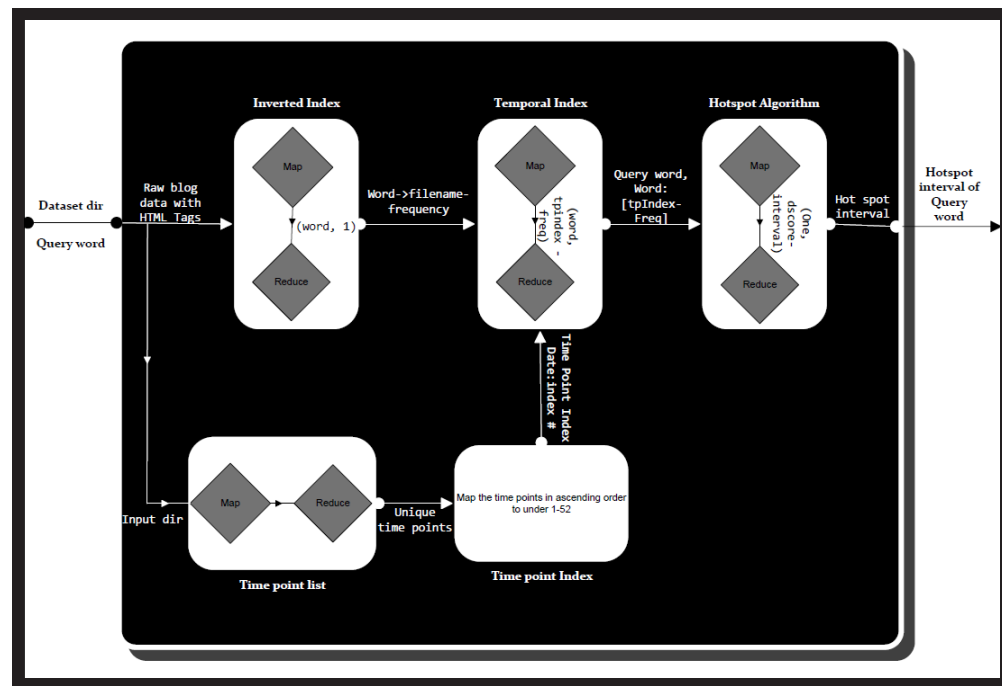


Figure 4.1 - The block diagram of data flow across the implementation for this thesis.

4.1 Inverted Index

This program deals with the cleaning up of the garbage in the blog data, building the inverted index for the dataset. The input to the program is all the contents of all the files in the data set and the output is the inverted index of the form word->filename-freq.

INPUT: Document Collection

```
hadoop@r1:~/aashokan/hadoop_inputs/Input10/NewSimpleInput
$ cat main and Erik Santos To Perform in Olympic Festival Closing Ceremony

<p>Our very own <strong>Erik Santos</strong> is among the chosen few across Asia to perform in the closing ceremony of the Olympic in Beijing.</p><p>As a representative of the Philippines, Erik will perform his signature song &#34;I Believe I Can Fly&#34;. This is the Moment.&#34;</p><p>Aside from Erik, the stars who will also perform in the said event are Asian Heartthrob <strong>Rain</strong> from Korea, the all-girl J-pop group <strong>Morning Musume</strong> from Japan and other artists from Hong Kong, Si Malaysia and Indonesia.<p>The said event is going to be held on September 18, 2008 at the incredible landmark structure China Centre for Cultural and International Exchange (CCTV) in Beijing.<p>It feels great to be invited to this prestigious event. It is an honor to represent the Philippines. I will best to perform well and make our country proud. Filipinos are some of the best performers in the world. Just performing alongside other Asian artists is an honor and a great privilege.<p>with WP-Autoblog (http://elliottback.com) -->
```

Figure 4.1.1. Example of a document in the document collection

OUTPUT: Inverted Index file of the form word :filename-freq

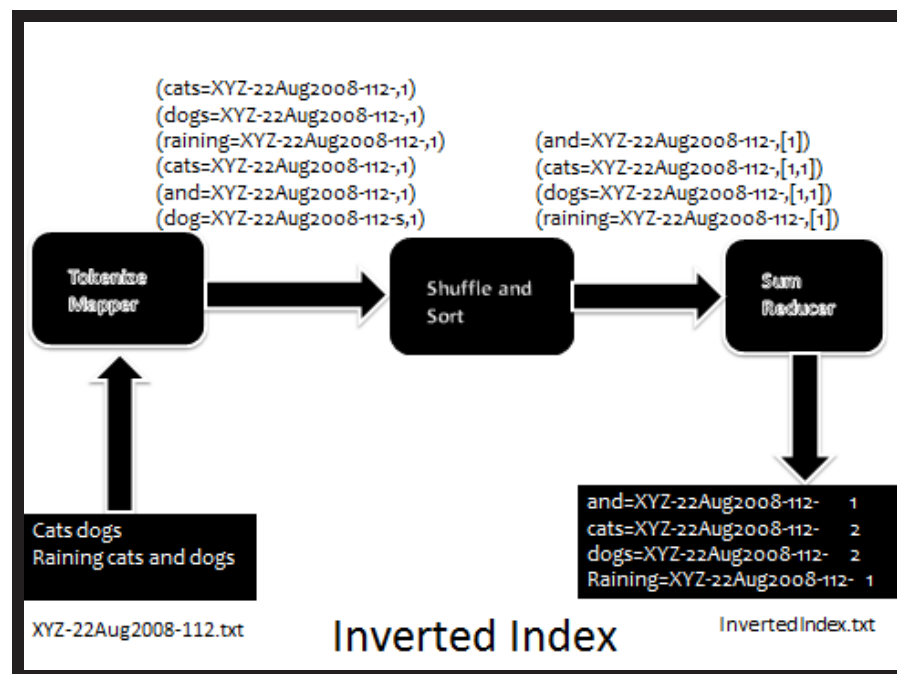
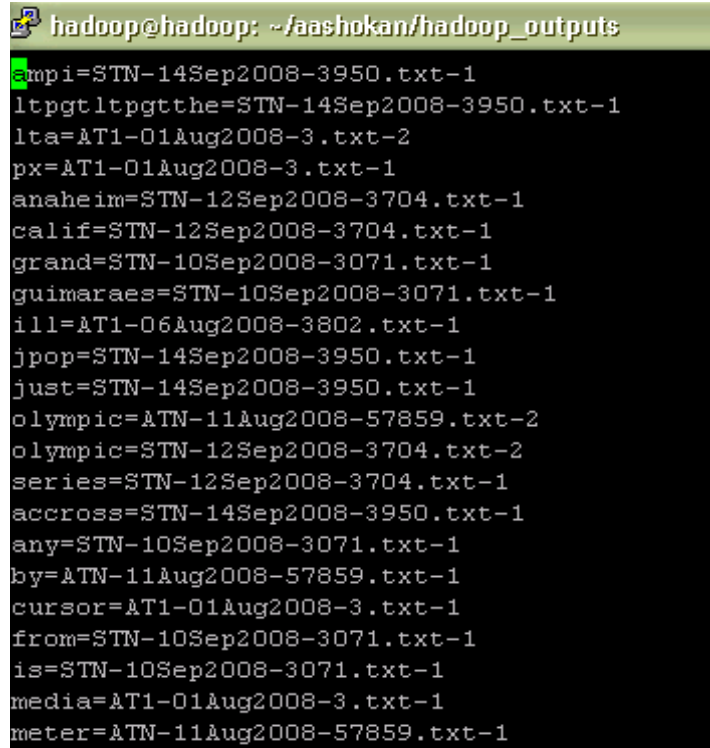


Figure 4.1.2 - Block diagram of Inverted Index Algorithm.



```

hadoop@hadoop: ~/aashokan/hadoop_outputs
ampi=STN-14Sep2008-3950.txt-1
ltpgtltpgtthe=STN-14Sep2008-3950.txt-1
lta=AT1-01Aug2008-3.txt-2
px=AT1-01Aug2008-3.txt-1
anaheim=STN-12Sep2008-3704.txt-1
calif=STN-12Sep2008-3704.txt-1
grand=STN-10Sep2008-3071.txt-1
guimaraes=STN-10Sep2008-3071.txt-1
ill=AT1-06Aug2008-3802.txt-1
jpop=STN-14Sep2008-3950.txt-1
just=STN-14Sep2008-3950.txt-1
olympic=ATN-11Aug2008-57859.txt-2
olympic=STN-12Sep2008-3704.txt-2
series=STN-12Sep2008-3704.txt-1
accross=STN-14Sep2008-3950.txt-1
any=STN-10Sep2008-3071.txt-1
by=ATN-11Aug2008-57859.txt-1
cursor=AT1-01Aug2008-3.txt-1
from=STN-10Sep2008-3071.txt-1
is=STN-10Sep2008-3071.txt-1
media=AT1-01Aug2008-3.txt-1
meter=ATN-11Aug2008-57859.txt-1

```

Figure 4.1.3 Snapshot of Inverted Index from our project.

ALGORITHM:

- i. Tokenizing the text in the document collection
- ii. Discarding of the noise/garbage in the documents.
- iii. Extracting the file name
- iv. Building a Inverted Index file of the form “*word :filename_freq*” where *freq* is the frequency of the *word* in the *filename*

```

Class SequentialInvertedIndex
{
    main()
    {
        1. Initialize HashMap<String,HashMap>
           InvertedIndex.
        2. Initialize HashMap<String,Integer> FileIndex.
        3. Go through each file in the document
           collection
    }
}

```

```

4. For each file with filename f
  a. Tokenize the contents of the file into
    words
  b. For every word w
    i. Remove all special characters from
       w
    ii. if length of w > 0 and length of w
        < 15
        if w does not exist in
        InvertedIndex
            Insert <w,<f,1>> to
        InvertedIndex
        else
            add 1 to the frequency of w
            in file name f.

    }
  }

Class MapReduceInvertedIndex
{
  map(files in dataset)
  {
    for every file fname
      tokenize the contents of fname
      for each word w in the tokenized file
        remove all special characters in w
        if w > 0 and w < 15
          emit(w->fname-,1)
  }

  reduce(w->fname-,[freqList])
  {
    for f:freqList
    {
      sum = sum + f;
    }
    emit(w->fname-,sum)
  }
}

```

4.2 Time Point List

The filenames in the data set are of the format “ABC-DateMonthYear-123.txt”. We extract the list of all time points in the data set using this program. The input to the program is the path to the directory containing the dataset and the output is a list of unique time points spanning the dataset. This is extracted from the filenames in the data set.

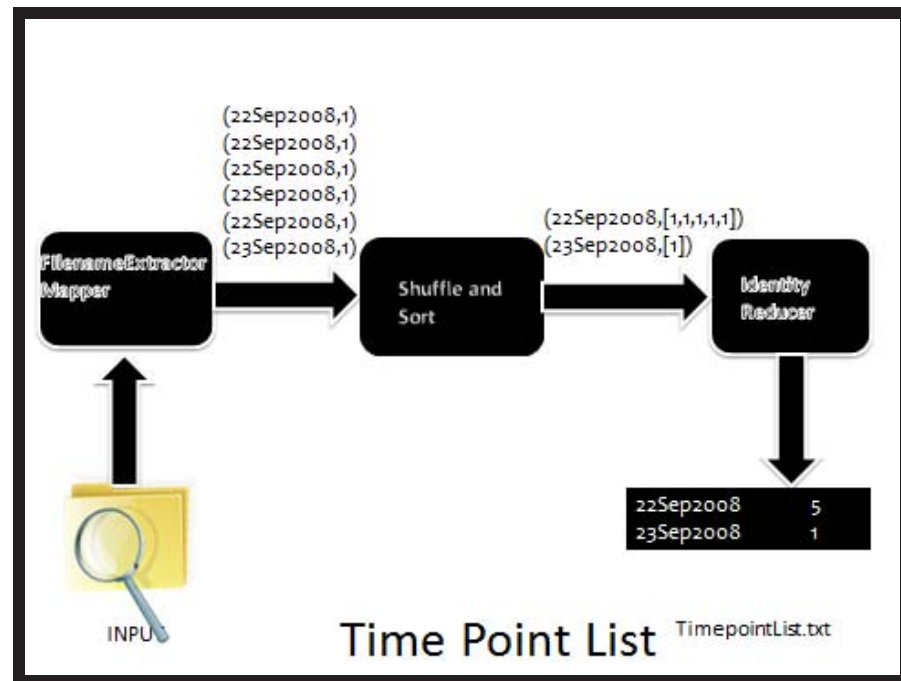


Figure 4.2.1 Block diagram of Time Point List Algorithm.

ALGORITHM:

- i. Read all the filenames
- ii. For each filename fname
 - a. Extracting the date from fname
- iii. Write it to a file

```
Class SequentialTimePointExtraction
```

```
{
    main()
    {
        1. Go through each filename in the data set directory
        2. For each file with filename fname
            a. Date = string in between - and -
            b. Write it to a file.

    }
}
```

```
Class MapReduceTimePointExtraction
```

```
{
    map(path to the dataset)
    {
        for every filename fname
            Date = string in between - and -
            emit(Date, 1)
    }

    reduce(Date, [1,1,1,1...flist])
    {
        no_of_files_for_date = 0;
        for f:flist
        {
            no_of_files_for_date = no_of_files_for_date
+ f;
        }
        emit(Date, no_of_files_for_date*);
    }
}
```

Note:- *The no_of_files_for_date values are a side effect of the program and we don't use it anywhere else.

4.3 TimePoint Index

This is a sequential java program where the unique sorted dates from the previous program are assigned a numerical value as a reference. The input to the program is the output list of unique time points from the TimePoint List program and the output list is a TimePoint Index for the date of the form date:Index number.

Eg.
 01Aug2008:1
 02Aug2008: 2
 .
 .
 .
 01Sept2008:32
 02Sept2008:33
 .
 .
 etc

ALGORITHM:

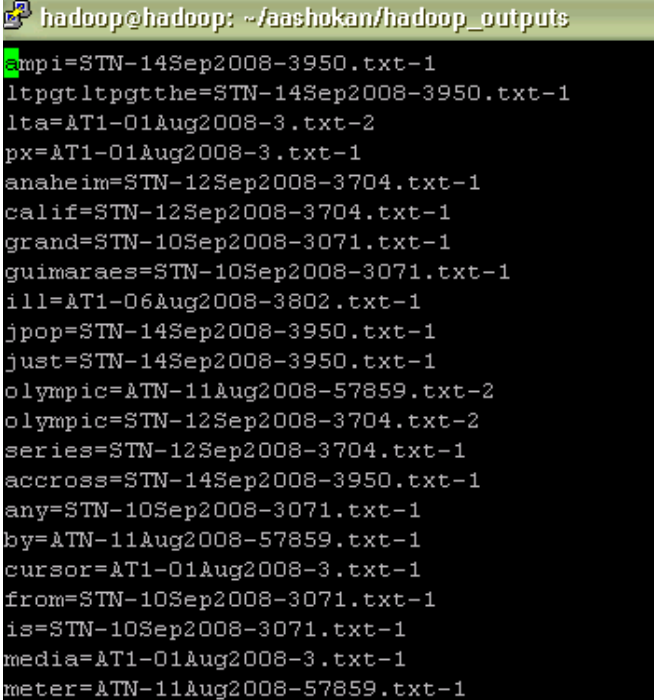
1. Open TimePointList.txt
2. Read each line and sort it in chronological order.
3. Assign the number 1-52 as indices to all the dates where 1 represents a date that precedes the date 2 represents.

4.4 Temporal Index

This is a key step in this experiment since the output of this step, the temporal index will be the input to the Hot spot extraction algorithm. This program used the Inverted Index and Time Point Index, to generate the Temporal Index file which we call the *TemporalIndex.txt* and it is of the form *word:[timepointindex-freq_in_the_timepoint]*
 - this way there will be exactly one entry for a word in the Temporal Index File.

INPUT: *InvertedIndex.txt* and *TimePointIndex.txt*

Inverted Index.txt

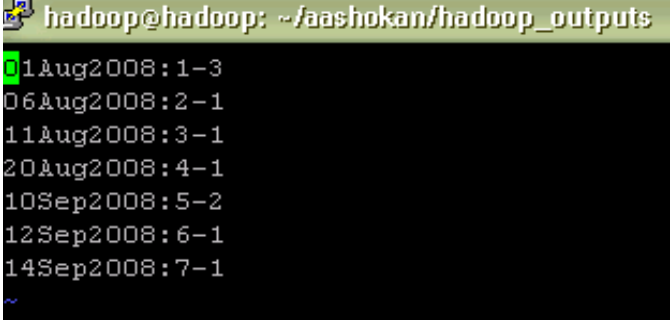


```

hadoop@hadoop: ~/aashokan/hadoop_outputs
ampi=STN-14Sep2008-3950.txt-1
ltpgtltpgtthe=STN-14Sep2008-3950.txt-1
lta=AT1-01Aug2008-3.txt-2
px=AT1-01Aug2008-3.txt-1
anaheim=STN-12Sep2008-3704.txt-1
calif=STN-12Sep2008-3704.txt-1
grand=STN-10Sep2008-3071.txt-1
guimaraes=STN-10Sep2008-3071.txt-1
ill=AT1-06Aug2008-3802.txt-1
jpop=STN-14Sep2008-3950.txt-1
just=STN-14Sep2008-3950.txt-1
olympic=ATN-11Aug2008-57859.txt-2
olympic=STN-12Sep2008-3704.txt-2
series=STN-12Sep2008-3704.txt-1
accross=STN-14Sep2008-3950.txt-1
any=STN-10Sep2008-3071.txt-1
by=ATN-11Aug2008-57859.txt-1
cursor=AT1-01Aug2008-3.txt-1
from=STN-10Sep2008-3071.txt-1
is=STN-10Sep2008-3071.txt-1
media=AT1-01Aug2008-3.txt-1
meter=ATN-11Aug2008-57859.txt-1

```

Figure 4.4.1- Example of *InvertedIndex.txt* – an input to Temporal Index Algorithm



```

hadoop@hadoop: ~/aashokan/hadoop_outputs
01Aug2008:1-3
06Aug2008:2-1
11Aug2008:3-1
20Aug2008:4-1
10Sep2008:5-2
12Sep2008:6-1
14Sep2008:7-1
~

```

Figure 4.4.2- Example of *TemporalIndex.txt* – an input to Temporal Index Algorithm

OUTPUT: *TemporallIndex.txt*, of the form *word:[timepointindex-
freq_in_the_timepoint]*


```

auto:: 1-1
but:: 5-1,1-1
easily:: 3-1
erik:: 7-1,7-2
full:: 5-1
games:: 2-1,3-1
ltpgtltptgtthe:: 7-1
on:: 3-1,7-1
perform:: 7-4,7-1
philippines:: 7-1,7-1
privilege:: 7-1
september:: 6-1,7-1
singapore:: 1-1
stars:: 7-1
style:: 1-1
summer:: 3-1
the:: 1-1,2-5,5-2,1-1,6-2,6-1,5-1,4-2,3-6,1-3,7-13
today:: 2-1,2-1,2-1
amphis:: 7-1
display:: 1-1
gold:: 1-1,6-1,6-1
his:: 3-1,7-1,1-1
hong:: 7-1
illusions:: 5-1
marksmanship:: 1-1
monday:: 3-1
sunday:: 3-1

```

Figure 4.4.3- Example of TemporalIndex.txt - the output of Temporal Index Algorithm.

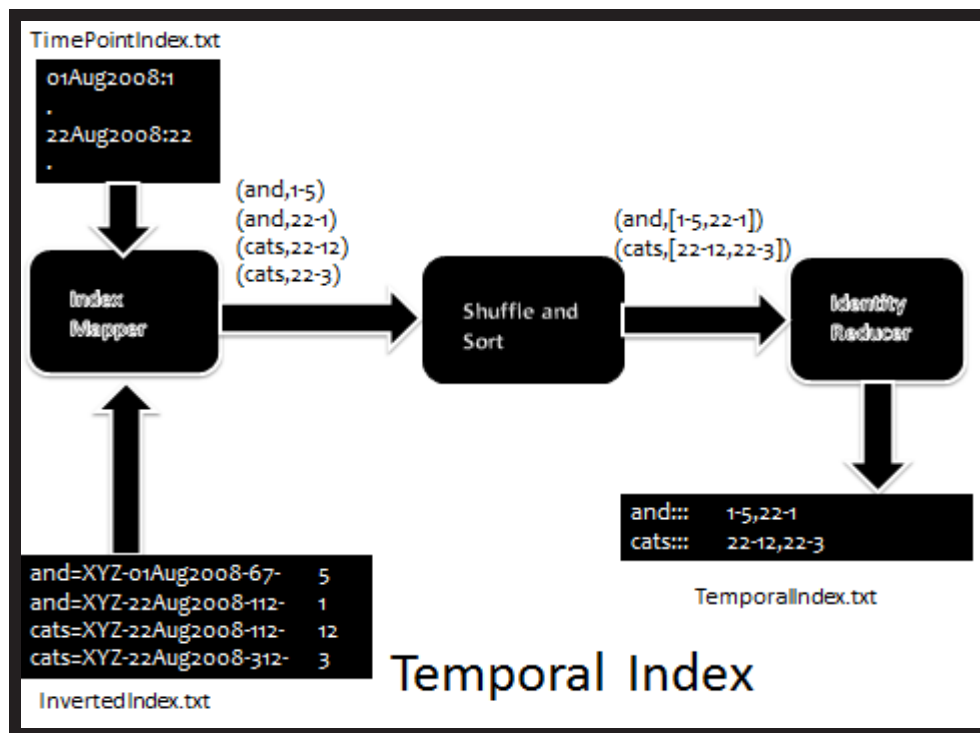


Figure 4.4.4- Block Diagram of Temporal Index Algorithm

ALGORITHM:

- Read the TimePoint Index file.
- Read the Inverted Index file
- For every line *ln* in the Inverted Index file

Replace the Date in the ln with the TimePoint index number corresponding to the date.

```
Class SequentialTemporalIndex
```

```
{
    main()
    {
        For every line iiln in Inverted Index
            Extract Date from ln
            For every line tiln in TimePoint Index
                Find the index for Date
                Replace Date in iiln with index for Date

    }
}
```

```
Class MapReduceTemporalIndex
```

```
{
    map(inverted index, timepoint index)
    {
        Read timepoint index and store it in a hashmap.
        for every line l in inverted index
            word = keyword from l
            tpointFreq = l - word
            date = extract date from tpointFreq
            tpindex = hashmap.get(date)
            replace date in tpointFreq with tpindex
            emit(word, tpointFreq)

    }

    reduce(word, [tpointFreq])
    {
        emit(word, [tpointFreq]);
    }
}
```

4.5 Hot Spot Computation

This is the heart of this experiment and talks about the implementation of the naïve hot spot extraction algorithm the sequential way and the MapReduce way.

INPUT: Word, *TemporalIndex.txt*

OUTPUT: Hot spot for the word, i.e. the time point at which the word has maximum occurrence.

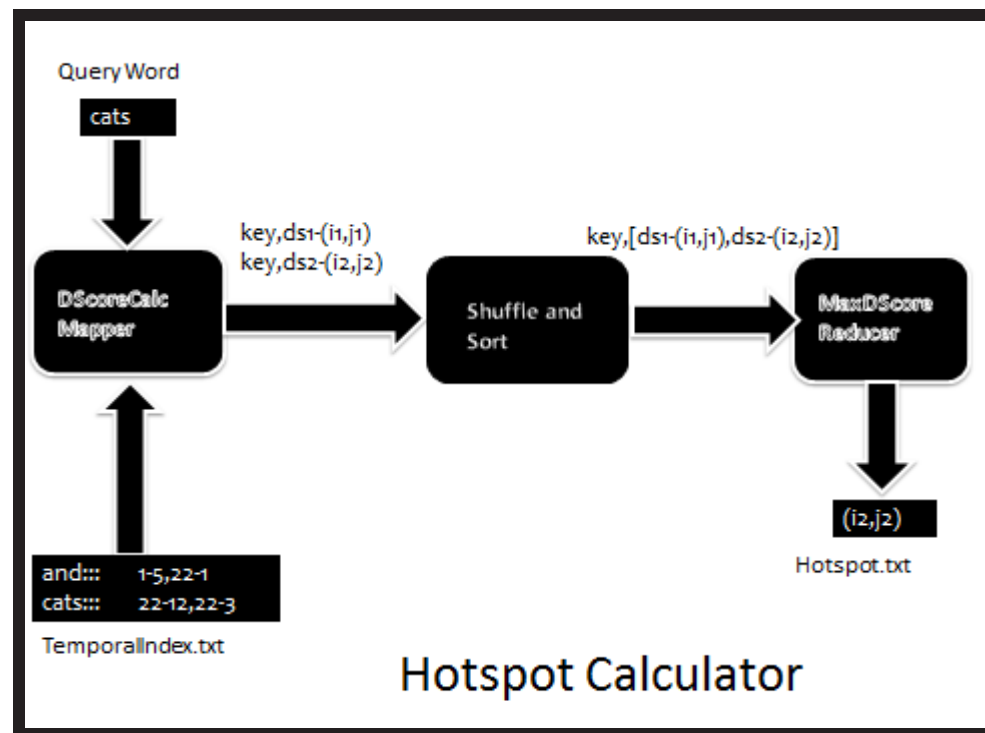


Figure 4.5.1– Block Diagram of Hotspot Calculator Algorithm

ALGORITHM:

1. Reading a word from the user whose hotspot is to be found.
2. Calculating the discrepancy score of the word for every possible interval using the formulae and the Temporal Index File
3. Calculating the m, M, b and B for a word
 4. Plugging it onto the formulae to calculate the discrepancy score

5. Remembering and returning the time points with the maximum discrepancy score
6. Returning the hot spot for the word.

Calculation of discrepancy score of an interval:

A time point is an instance of time with a given base granularity, such as a second, minute, day, month, year, etc. A time point can be represented by a single numerical value, specifying a given second, minute, day, etc. A time period T is a sequence of n time point's $t_1 \dots t_n$. An interval T_{ij} ($1 \leq i \leq j \leq n$) of T is a sequence of consecutive time points, starting at time point t_i and ending at time point t_j , in T . Let T_{ij} and T_{kl} be two intervals of T . T_{ij} is contained in T_{kl} if $i \geq k$ and $j \leq l$ [3].

Suppose our dataset spans across a time period T . Discrepancy score of topic during a time interval T_{ij} of T is calculated by comparing its presence during T_{ij} to its presence in the rest of the time period. Let m be the number of times the topic appears in T_{ij} , and b be the total number of times all of the topics appear in T_{ij} . In addition, let M denote the number of times the topic appears in the entire time period T ; and B denote the number of time all topics appear during T . We calculate the discrepancy score of p in t using the following formula:

$$d(m, b) = m \log \frac{\frac{m}{b} + (M - m) \log(M - m)}{B - b}, \text{ if } \frac{m}{b} \geq \frac{M}{B}$$

where,

m is the frequency of w in the interval (i, j)

b is total frequency of all words in interval (i, j)

M is the total frequency of w in the document collection N

B is the total frequency of all words in the document collection N

Class SequentialHotSpotAlgorithm

```
{
    main()
    1. If  $n$  is the # of time points in document collection
    2.  $d_{max} = -\infty$ 
    3. for  $i = 1$  to  $n$ 
        i. for  $j = i$  to  $n$ 
            I.  $m_{ij} = b_{ij} = 0$ 
            II. for  $k = i$  to  $j$ 
                A.  $m_{ij} = m_{ij} + m_k$ 
                B.  $b_{ij} = b_{ij} + b_k$ 
            III. Compute  $d_{ij}$  for  $(i,j)$ 
            IV. if( $d_{ij} \geq d_{max}$ )
                A.  $d_{max} = d_{ij}$ 
    4. hotspot =  $(i,j)$ 
}
```

Class MapReduceHotSpotExtraction

```
{
    map(temporal_index file, query_word)
    {
        dmax = negative infinity
        B = Calculate B from temporal_index file
        M = Calculate M from temporal_index file for the
query_word
        for i: 1 to N
        {
            for j: i to N
            {
                for k: i to j
                {
                    m = calculate m(i,j) for
query_word
                    b = calculate b(i,j) for
query_word
                }
                d(i,j) = calculate discrepancy score
            }
            for interval(i,j)
            {
                if( $d(i,j) > d_{max}$ )
                {
                    dmax = d(i,j)
                    dinterval = (i,j)
                }
            }
        }
    }
}
```

```

        }
    }
    emit(any_key,dmax+"-"+dinterval)
}

}

reduce(any_key,list of dmax-dinterval)
{
    hotspotDS = negative infinity
    hotspotInt = null;
    for dScoreInt : list of dmax-dinterval
    {
        dScore =
dScoreInt.substring(0,dScoreInt.indexOf("-"))
        dInt =
dScoreInt.substring(dScoreInt.indexOf("("),dScoreInt.indexOf
(""));
        if(dScore > dScoremax)
        {
            hotspotDS = dScore
            hotspotInt = dInt
        }
    }
    emit(hotspotInt, hotspotDS)

}

}

```

5. Experiments

5.1 Dataset

Our dataset came from the Spinn3r which provides high volumes of fresh data, tapping you into worldwide conversation. Spinn3r is a web service for indexing the blogosphere. It provide raw access to every blog post being published - in real time. Spinn3r handles all the difficult tasks of running a spider/crawler including spam prevention, language categorization, ping indexing, and trust ranking [11].

Spinn3r was founded in late 2005 by web crawler and RSS expert Kevin Burton. Mr. Burton is a serial entrepreneur and sold his previous company, *Rojo* to *Six Apart* in late 2006. Spinn3r was originally built to power *Tailrank*, a real-time blog analysis and topical relevance index which launched in early 2006. The architecture behind Spinn3r was influenced by two large projects. One was *Rojo*, which had a 500GB-1TB search index. The other was *NewsMonster*, one of the first and still the most advanced client side aggregator, with a high performance crawler integrated at its core [11].

Spinn3r was launched in August 2007 as a dedicated product after having numerous requests to license its backend infrastructure. Since launching, Spinn3r has been consistently adopted by new startups needing access to the blogosphere. Spinner is now providing crawl infrastructure for startups as well as dozens of universities and hundreds of researchers [11].

Our dataset is a small subset of the Spinn3r blog data released for analysis purposes to researchers. It spans from August 2008 – September 2008. The dataset consist of around 20,000 files of blog data from and is approximately 200MB in size. Each time point we have identified is a single day and we have 52 such time points across the entire dataset.

5.2 Machine Configuration

The run environment consisted of an 8 – node cluster. The specifications of the cluster are as follows:

- 8 x Dell PowerEdge R410 servers. Each of these servers have the following:

- Hardware:

- * 4 x 6-core CPUs (Intel Xeon X5660 @ 2.80 GHz)

- * 128GB of memory

- * DFS storage: 200GB per node (Total: 1.4 TB for the cluster)

- * 1 Gigabit NIC

- Software:

- * 64-bit Ubuntu Linux 10.04.4 LTS edition

- * Hadoop version: 0.20.203.0 with a built in Java version 1.6.0_26

5.3 Results

5.3.1 Summary

- Map Reduce algorithm can be significantly more efficient than sequential algorithm in handling computations on large data sets.

- As the research shows, Map Reduce algorithms have significant potential in several areas of computing including emerging fields such as analysis of hot spots.
- By and large, computing world has not utilized Map Reduce algorithms for hot spots, and the results of the experiments show that the algorithm's efficiency and scalability is well suited to hot spot based real world analysis.

5.3.2 Inverted Index and Temporal Index Sizes

The blog data was partitioned into one-week chunks and the hotspot computation was run on partitioned data for 1 week through 8 weeks to study the computation times and other patterns. The size of the inverted index went from approximately 10kB to 114kB as we included all 52-time points. The table in Figure 5.3.2.1 lists the size of the inverted index file and temporal index file for data set.

Time Frame	Inverted Index(kB)	Temporal Index(kB)
1 week	10187	1671
2 weeks	39094	6500
3 weeks	67202	11066
4 week	93529	15208
5 weeks	102830	16695
6 weeks	108460	17621
7 weeks	112977	18355
8 weeks	114541	18610

Figure 5.3.2.1 – This tables shows the increase in the size of Inverted Index and Temporal Index in kB as the dataset grew from 1 week of data to 8 weeks of data.

The graph in Figure 5.3.2.2 below shows the increase in the size of the temporal index as we increased the dataset by one week at a time. The x-axis shows the increase in dataset as we increment by a week at a time.

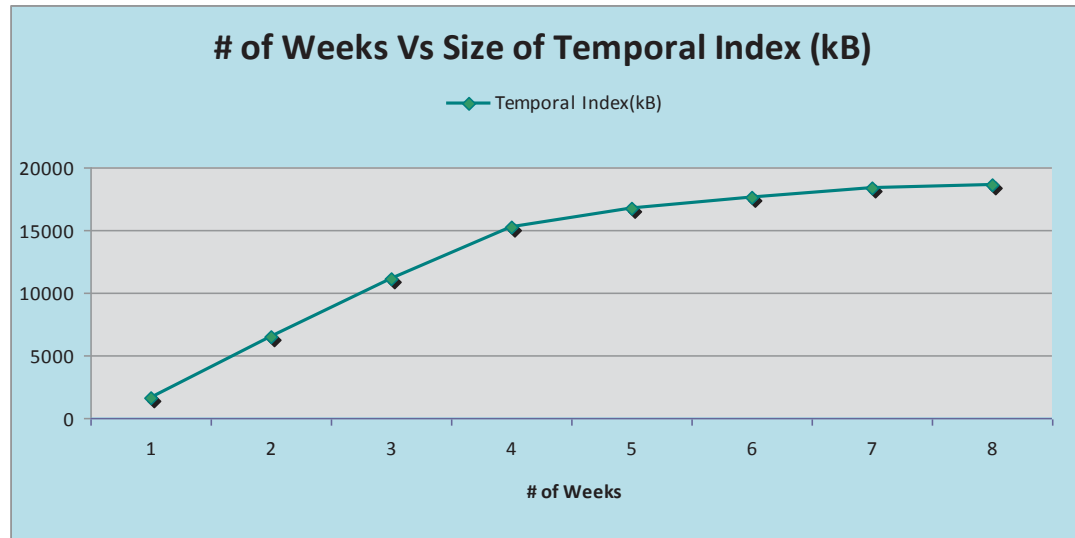


Figure 5.3.2.2 – This graph shows the trend of Temporal Index in kB as the dataset grew from 1 week of data to 8 weeks of data.

5.3.3 Keyword Count

The total number of distinct keywords in the temporal index also went up as we increased the dataset a week at a time. The table in Figure 5.3.3.1 shows list the number of distinct keyword contained in each of the temporal index.

Time Frame	# of keywords	Temporal Index(kB)
1 week	41503	1671
2 weeks	106486	6500
3 weeks	148207	11066
4 week	179037	15208
5 weeks	190398	16695
6 weeks	197299	17621
7 weeks	202447	18355
8 weeks	204283	18610

Figure 5.3.3.1 – This tables shows the increase in the # of keywords and the size of Temporal Index in kB as the dataset grew from 1 week of data to 8 weeks of data.

Figure 5.3.3.2 is a graph that shows the increase in the number of distinct keywords as the size of the dataset was increased one week at a time. The x-axis depicts the incremental increase in dataset while the y-axis gives the total distinct keywords.

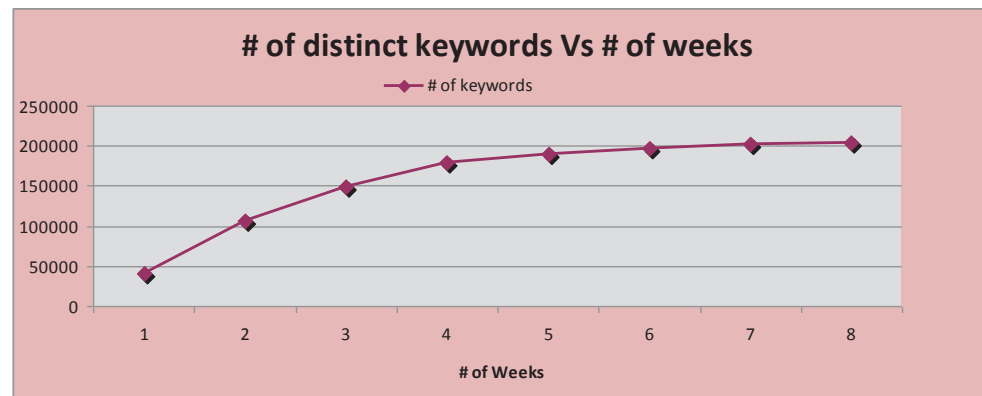


Figure 5.3.3.2 – This graph shows how the number of distinct keywords in the dataset grew across 1 week of data to 8 weeks of data.

5.3.4 Hotspots for some interesting keywords

The MapReduce version of the hotspot computation was run for a number of keywords with the largest dataset, i.e. the dataset consisting of data from 1st August 2008 to 30th Sept 2008. Each run took an average of 1 hour and 46 minutes of execution time.

Table in Figure 5.3.4.1 shows the hotspot interval obtained for all of the interesting keywords from the runs. The results of this runs was found to be interesting for two reasons:-

1. The hotspots of related keywords overlap. Eg. *policies* and *congress* both share a common interval window of 1st August 2008 to 6th August 2008, *mortgage* and *recession* has perfectly aligned hotspot interval of a single day, 14th August 2008

and so on as can be seen in the table shown in Figure 5.3.4.1. Also the Figure 5.3.4.2 is a timeline graph depicting the hotspot intervals of these keywords with the hotspot interval in the x-axis and the keyword in the y-axis.

2. The above observation justifies the meaningfulness and usefulness of hotspot computation.

Finding related keywords or temporal synonyms is an important research topic, especially the synonyms that change over time. The study of temporal synonyms is an altogether different topic and is beyond the scope of this project. However, the MapReduce style computation of hotspots can certainly be used in scaling the temporal synonym extraction problem.

Keyword	Start Interval	End Interval
policies	1-Aug-08	9-Aug-08
congress	1-Aug-08	6-Aug-08
yahoo	4-Aug-08	5-Aug-08
telemundo	5-Aug-08	10-Aug-08
nbcolympics	10-Aug-08	10-Aug-08
mortgage	14-Aug-08	14-Aug-08
recession	14-Aug-08	14-Aug-08
economy	23-Aug-08	23-Aug-08
stocks	23-Aug-08	23-Aug-08
democrats	27-Aug-08	27-Aug-08
clinton	27-Aug-08	17-Sep-08
exports	27-Aug-08	27-Aug-08
euros	28-Aug-08	21-Sep-08
hiring	29-Aug-08	29-Aug-08
republicans	30-Aug-08	21-Sep-08
summer	30-Aug-08	31-Aug-08
elections	10-Sep-08	10-Sep-08
unemployment	18-Sep-08	19-Sep-08
banking	24-Sep-08	25-Sep-08

Figure 5.3.4.1 – Shows the Hotspot interval of some interesting keywords found in our dataset.

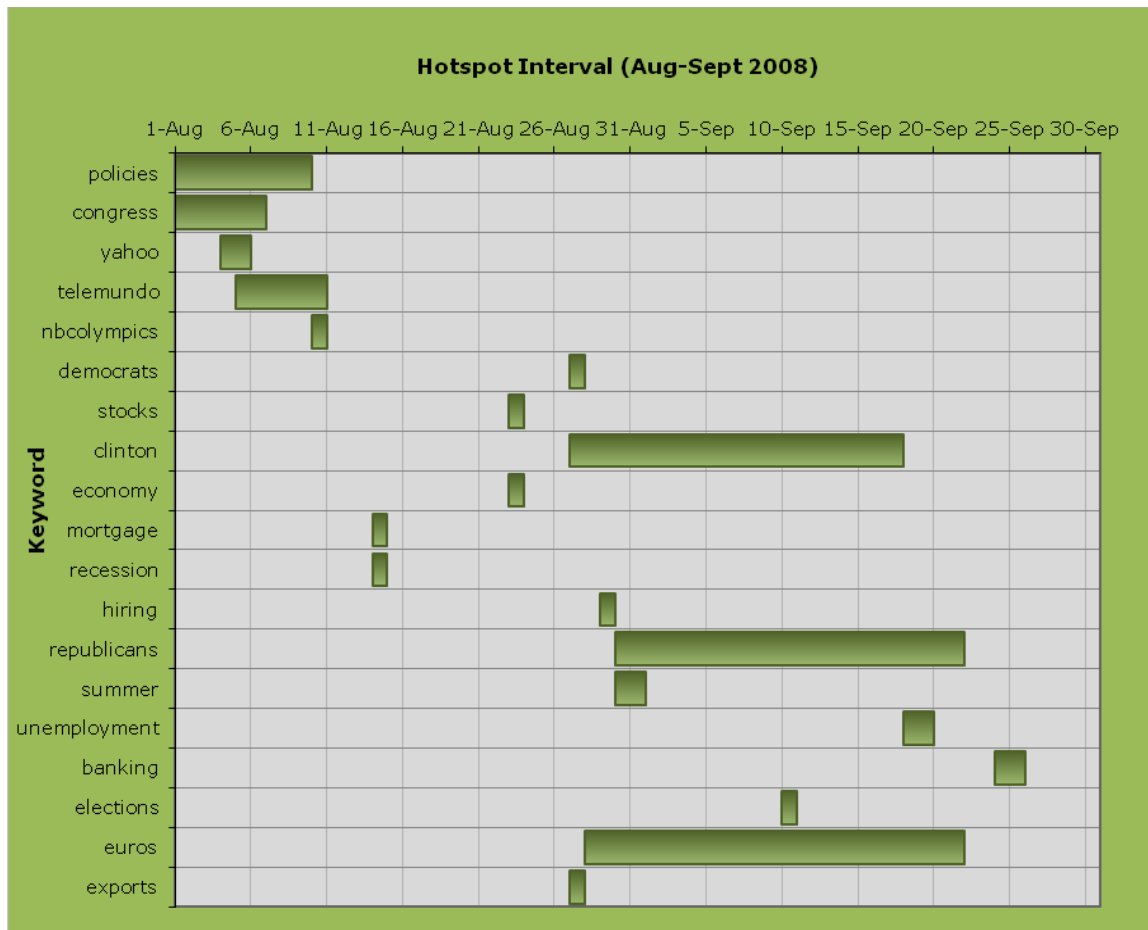


Figure 5.3.4.2 – Hotspot Timeline graph showing the hotspot intervals of some interesting keyword in our dataset.

5.3.5 Comparison of execution time and size of temporal index against growing dataset

It has been observed that the MapReduce execution time does not increase as rapidly as the increase in the dataset as the dataset grows. Table in Figure 5.3.5.1 is a comparison of the rate of change of the Map Reduce execution time with the increase in size of the temporal index. The graph in Figure 5.3.5.2 plots this pattern with the x-axis showing the increase in dataset and the y-axis on the left showing the MapReduce execution time in seconds and y-axis on the right showing the size of temporal index in kB.

Time Frame	Size of Temporal Index	MapReduce Execution Time (secs)
1 Week	1671	42
2 Weeks	6500	102
3 Weeks	11066	332
4 Weeks	15208	966
5 Weeks	16695	1975
6 Weeks	17621	3232
7 Weeks	18355	5119
8 Weeks	18610	6208

Figure 5.3.5.1 – MapReduce execution time as the size of temporal index increases.

Figure 5.3.5.3 depicts a graph showing the rate of change of MapReduce execution time as the size of temporal index increases. The x-axis shows the incremental increase in dataset while the left y-axis shows size of temporal index in kB and right y-axis shows the % increase in MapReduce execution time.

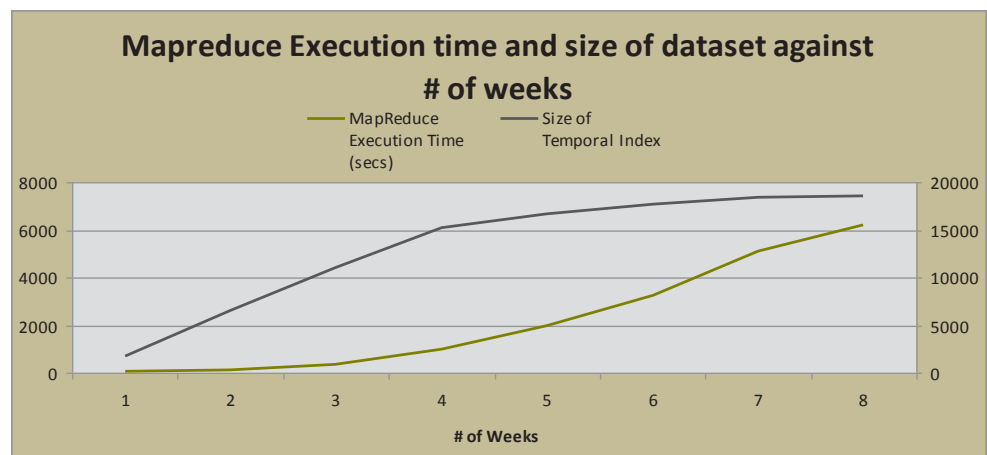


Figure 5.3.5.2 – Increase in the trend of MapReduce execution time and the Temporal Index size as the dataset increases.

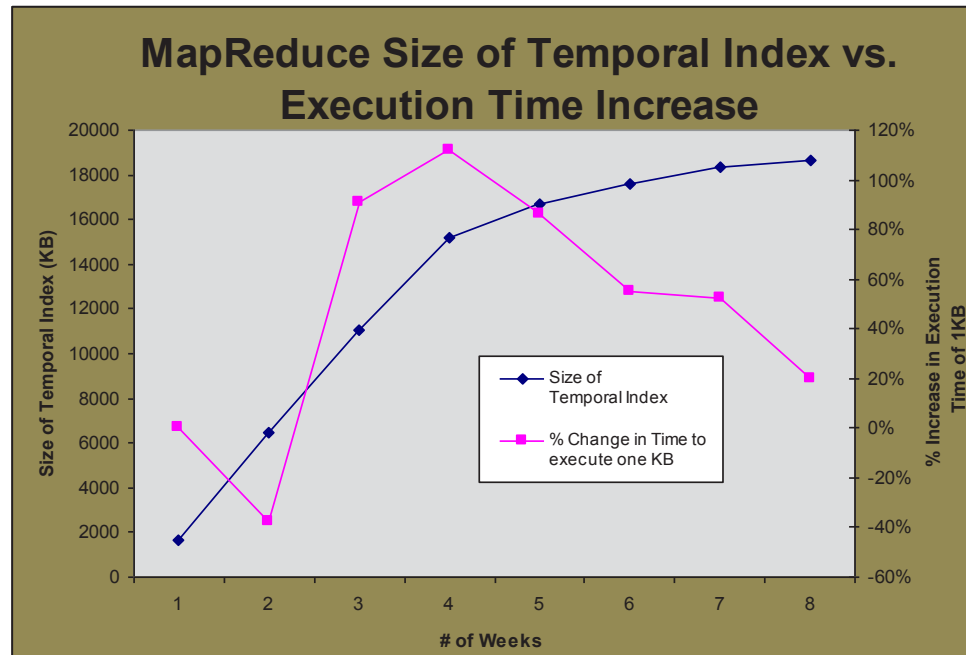


Figure 5.3.5.3 – Rate of change of increase in the MapReduce execution time as the size of Temporal Index increases.

5.3.6 MapReduce Implementation Vs Sequential Implementation

Even though the Map Reduce execution time is more than the Sequential execution time for the first run, as the size of the dataset grows the MapReduce time is much better than the Sequential time. For our largest dataset the Map Reduce time is 2.64 times better than its corresponding Sequential time and it can be projected that the Map Reduce time would continue to improve as the dataset increases further. Table in Figure 5.3.6.1 shows the hotspot interval obtained for the keyword *yahoo* for various runs as the dataset incrementally increased by a week at a time.

The graph in Figure 5.2.6.2 plots is a comparison of the MapReduce execution time vs the Sequential execution time for finding the hotspot of the keyword *yahoo*. The x-axis shows the incremental increase in size of the dataset while the y-axis shows the execution time in seconds.

Time Frame	Run #	HotSpot	MapReduce Execution Time (secs)	Sequential Execution Time (secs)
1 Week	Run1	04Aug2008 to 05Aug2008	42	8
2 Weeks	Run2	04Aug2008 to 05Aug2008	102	136
3 Weeks	Run3	04Aug2008 to 05Aug2008	332	727
4 Weeks	Run4	04Aug2008 to 05Aug2008	966	2220
5 Weeks	Run5	04Aug2008 to 05Aug2008	1975	4665
6 Weeks	Run6	04Aug2008 to 05Aug2008	3232	8119
7 Weeks	Run7	04Aug2008 to 05Aug2008	5119	13281
8 Weeks	Run8	04Aug2008 to 05Aug2008	6208	16427

Figure 5.3.6.1 – Shows the Hotspot interval, MapReduce execution time and Sequential Execution time for the keyword “yahoo”.

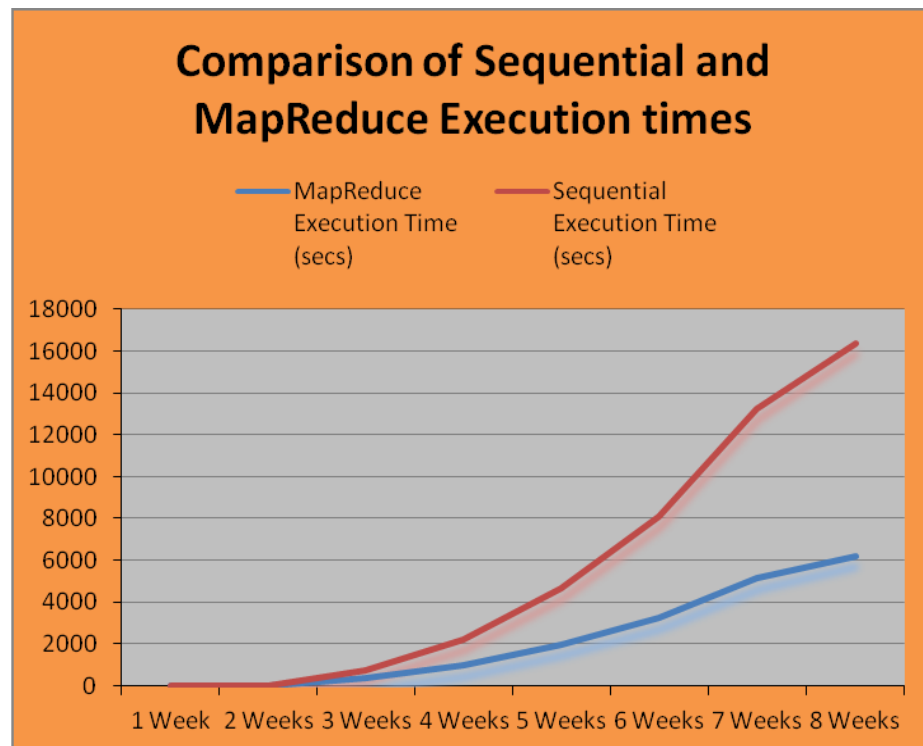


Figure 5.3.6.2 – Tread of Sequential Execution time and MapReduce Execution time for the keyword “yahoo”. MapReduce execution time performs 2.64 times better than Sequential execution time for the largest dataset.

6 Conclusion and Future work

In this thesis, we mainly address the challenges of using the MapReduce model to scale the hot spot extraction algorithm. We described the algorithm design and implementation on Hadoop. The scalability and performance of the implementation were investigated. Implementations were done in Java.

Implementing the Hotspot Algorithm in the MapReduce involved doing some pre-processing to clean up the data, creation of inverted index, time point index and temporal index, which would ultimately be the input to the Hotspot Algorithm. Re-implementing an existing algorithm in Map Reduce framework involves a complete rethinking of the problem and this was one of the most challenging parts of the project.

MapReduce abstracts most of the data handling logic for us which can be challenging as the requirement to keep track of the data distributions increase. While abstraction in MapReduce allows for working understanding of distributed and parallel programming sufficient for program development, debugging, fine tuning, and documentation gets proportionally difficult.

As captured in the results, as the data continues to grow, the MapReduce algorithm performs exponentially better than sequential algorithms, and this efficiency is expected to improve significantly for even larger datasets. One area for future work involves running the existing MapReduce algorithm for a much larger dataset to confirm the hypothesis.

During the experiments, it was also observed that the Mapper had longer execution times than the reducer and it was attributed to the $O(n^2)$ algorithm the Mapper was running. The second area for future work involves implementing an improved version (EHE) of the hotspot algorithm referenced in [3] in the MapReduce model.

A third area for future work involves re-engineering the current algorithm to calculate hot spot for streaming data which would provide a good test on the scalability and efficiency of the algorithm given that streaming data will be dynamic and could grow quickly in a short time period.

Bibliography

1. Wikipedia listing, <http://en.wikipedia.org/wiki/MapReduce>
2. O'Reilly Publication, *Hadoop: The Definitive Guide*.
3. Wei Chen, Parvathi Chundi: Extracting hot spots of topics from time-stamped documents. *Data Knowl. Eng.* 70(7): 642-660 (2011)
4. Jeffrey Dean and Sanjay Ghemawat, *Google Inc*: MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004)*, p.137-150, 2004, San Francisco, California.
5. Jimmy Lin: Exploring Large-Data Issues in the Curriculum: A Case Study with MapReduce. *Proceedings of the Third Workshop on Issues in Teaching Computational Linguistics (TeachCL-08)*, p.54-61, Columbus, Ohio, USA, June 2008.
6. Jimmy Lin, Chris Dyer: Data-Intensive Text Processing with MapReduce. *Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts* p.1-2
7. Internet listing, <http://binarynerd.com/java-tutorials/distributed-computing/hadoop-mapreduce-101.html>
8. Jerry Zhao, Jelena Pjesivac-Grbovic MapReduce: The programming model and practice, *SIGMETRICS(2009)*
9. Jimmy Lin: Exploring Large-Data Issues in the Curriculum: A Case Study with MapReduce

-
10. Jonathan Cohen: Graph Twiddling in a MapReduce World. *Cloud Computing Journal, Computing in Science and Engineering, Volume 11 Issue 4, July 2009, p 29-41.*
 11. Website listing: <http://spinn3r.com/>
 12. Chen, Wei, and Parvathi Chundi: An approach for discovering hot spots of topics from time stamped documents. *2008 SDM Text Mining Workshop. 2008.*
 13. W. Chen, P. Chundi: Extracting hot spots of basic and complex topics from time stamped documents, *2009 IEEE Symposium on Computational Intelligence and Data Mining, 2009, pp. 125–132.*
 14. Wikipedia Listing:
http://en.wikipedia.org/wiki/Hadoop_Distributed_File_System#Hadoop_Distributed_File_System
 15. Scaling Genetic Algorithms Using MapReduce: *ISDA '09 Proceedings of the 2009 Ninth International Conference of Intelligent Systems Design and Applications, Pages 13-18.*
 16. Ricardo Baeza-Yates, Berthier Ribeiro-Neto: Modern Information Retrieval. ACM Press Books 1999.
 17. Scan Statistics Website: <http://www.satscan.org/>
 18. Kulldorff M. Spatial scan statistics: Models, calculations and applications. In Balakrishnan and Glaz (eds), Recent Advances on Scan Statistics and Applications. Boston, USA: Birkhäuser, 1999. [online]

